Escuela
Politécnica
Superior

Dimensión fractal como descriptor 3D

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
José Francisco Domenech Gomis
Tutor/es:
Miguel Angel Cazorla Quevedo
Félix Escalona Moncholí

Junio 2020

Universitat d'Alacant
Universidad de Alicante

Dimensión fractal como descriptor 3D

Autor
José Francisco Domenech Gomis

Tutor/es
Miguel Angel Cazorla Quevedo
Departamento de Ciencia de la Computación e Inteligencia Artificial
Félix Escalona Moncholí
Departamento de Ciencia de la Computación e Inteligencia Artificial

Grado en Ingeniería Informática

Escuela
Politécnica
Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Junio 2020

Preámbulo

En este proyecto se va a investigar la aplicación de la dimensión fractal como descriptor global para el reconocimiento de objetos definidos por nubes de puntos en tres dimensiones. Se parte de la hipótesis de que la dimensión fractal puede ofrecer una medida de la rugosidad o irregularidad de una superficie y por ello podría llegar a ser usada como descriptor.

Para esto habrá que implementar por una parte el método de cálculo de la dimensión fractal, en este caso por el método de conteo de cajas (*box counting* en inglés) y por otra parte, con los resultados de esta probar varios clasificadores que nos permitan reconocer distintos objetos Tres Dimensiones (3D).

Agradecimientos

Para empezar, me gustaría agradecer a mis tutores, Félix y Miguel, su gran labor dirigiéndome a lo largo de estos nueve meses de trabajo, solucionando todos los problemas que han ido apareciendo y sobretodo apreciando el esfuerzo incluso cuando los resultados parecían no mejorar. Este agradecimiento se puede extender por supuesto a todos los buenos profesores con los que he tenido la suerte de coincidir a lo largo del grado.

Naturalmente, también me gustaría dar las gracias a mi familia, por apoyarme a lo largo de esta etapa en mis estudios y por inculcarme desde pequeño la curiosidad por aprender más, sin la que hoy en día no sería todo lo que soy.

Por último, agradecer el apoyo a todos mis amigos, tanto a los que han ido apareciendo en mi vida a lo largo de estos cuatro años, como a los que llevan ya alguno más formando una parte indispensable de ella.

Dedicado a todos los que de una forma u otra han sido parte de estos cuatro años de carrera.

Contrariamente a la opinión popular,
las matemáticas tratan sobre simplificar la vida,
no sobre complicarla.

Benoît Mandelbrot.

Índice general

1 Introducción **1**
1.1 Dimensión fractal 1

2 Marco Teórico **3**
2.1 Fractales 3
2.1.1 Dimensión fractal 4
2.1.2 Dimensión fractal por conteo de cajas 6
2.2 Clasificadores 8
2.2.1 K-Vecinos más cercanos 8
2.2.1.1 Distancia Euclídea 8
2.2.1.2 Distancia de Mahalanobis 9
2.2.2 Support Vector Machines 10
2.2.3 Redes neuronales 10
2.2.3.1 Perceptrón multicapa 12
2.2.3.2 Redes neuronales convolucionales 12
2.3 Conceptos de visión artificial 14
2.3.1 Formas de representar objetos 3D 15
2.3.2 Detectores 15
2.3.3 Descriptores 16
2.4 Trabajos previos y estado del arte 17
2.4.1 Reconocimiento 3D proyectando a 2D 17
2.4.2 Reconocimiento 3D sin proyecciones 19
2.4.3 Aplicación de la dimensión fractal 20

3 Objetivos y motivación **25**

4 Metodología **27**
4.1 Open3D 27
4.2 Dataset 27
4.2.1 ModelNet10 29
4.2.2 ModelNet40 29

5 Desarrollo **33**
5.1 Box counting 33
5.1.1 Lectura y transformación a nube de puntos 35
5.1.2 PointCloud 35
5.1.3 PointCloud.crop 35
5.1.4 VoxelGrid 35
5.1.5 Ajuste de rectas 36

5.2 KNN 37
5.3 Redes Neuronales 38
5.4 SVM 39

6 Experimentación **41**
6.1 Resultados dimensión fractal 41
6.2 KNN 44
6.2.1 Dimensión de la figura completa 44
6.2.2 Dimensión realizando subdivisiones 44
6.2.2.1 Distancia euclídea 45
6.2.2.2 Histogramas y su uso en clasificación 47
6.2.2.3 Distancia de Mahalanobis 47
6.3 Redes neuronales 51
6.3.1 Perceptrón multicapa 51
6.3.2 Redes densamente conectadas 52
6.3.3 Redes convolucionales 53
6.4 SVM 53
6.4.1 ModelNet40 53

7 Conclusiones **57**
7.1 Trabajos futuros 57

Bibliografía **59**

Lista de Acrónimos y Abreviaturas **63**

Índice de figuras

2.1 Ejemplos de fractales naturales. 4
2.2 Ejemplos de fractales puros. 5
2.3 Esquema resumen método *box counting* 7
2.4 Funcionamiento KNN 9
2.5 Funcionamiento SVM. 10
2.6 Esquema red neuronal típica. 11
2.7 Algunas funciones de activación. 13
2.8 Fronteras de decisión problema no linealmente separable. 13
2.9 Operación de colvolución. 14
2.10 Operación de *pooling*. 14
2.11 Comparativa entre nubes de puntos, *voxel grids* y mallas poligonales. 15
2.12 Comparativa de dos populares detectores 3D. 16
2.13 Comparativa de combinaciones entre detectores y descriptores. 16
2.14 Ejemplo descriptores. 17
2.15 Esquema de entrenamiento de RotationNet. 18
2.16 Ejemplo de como LonchaNet divide las figuras 19
2.17 Arquitectura LonchaNet 20
2.18 Ejemplo PointNet 21
2.19 Segmentación hojas para identificación de vegetales. 21
2.20 Relación entre urbanismo y dimensión fractal. 22
2.21 Dimensión fractal en pacientes de alzhéimer y sujetos sanos. 23

4.1 Error al crear VoxelGrid sobre malla poligonal 28
4.2 Comparativa clases *night_stand* y *dresser* 30
4.3 Comparativa clases *cup*, *plant*, *vase* y *flower_pot* 32

5.1 Aproximación por mínimos cuadrados 34
5.2 Ejemplo uniform sampling 36
5.3 Muestra de cómo se divide una figura. 37
5.4 Evolución del VoxelGrid resultante en el conteo de cajas. 38

6.1 Curva log-log pirámide de Sierpinski 41
6.2 Dimensión fractal sobre algunas figuras. 42
6.3 Resultados de dimensión fractal sobre algunos modelos del *dataset*. 43
6.4 Matriz de confusión figura completa KNN 44
6.5 Comparativa resultados K-Nearest Neighbours (KNN) para distinto número de subdivisiones y valores de K usando distancia euclídea. 45
6.6 Matriz de confusión realizando 5^3 subdivisiones usando KNN con $k = 1$ y distancia euclídea 46

6.7 Matriz de confusión clasificando por histogramas. 46
6.8 Histogramas de 20 y 100 intervalos y sus matrices de confusión. 48
6.9 Comparativa histogramas en función del número de divisiones de la figura. . . 49
6.10 Comparativa resultados KNN para distinto número de subdivisiones y valores de K usando distancia de Mahalanobis 50
6.11 Matriz de confusión realizando 8^3 subdivisiones usando KNN (distancia Mahalanobis) con $k = 1$ 50
6.12 Matriz de confusión realizando 9^3 subdivisiones usando red densamente conexa básica 51
6.13 Historiales de *accuracy* y *loss* antes y después de aplicar Dropout. 52
6.14 Matriz de confusión realizando 5^3 subdivisiones usando SVM 54
6.15 Acierto en ModelNet40 para distinto número de divisiones de la figura. 54
6.16 Matriz de confusión de ModelNet40 realizando 7^3 subdivisiones usando SVM 55

Índice de tablas

4.1 Tabla resumen ModelNet10 29
4.2 Tabla resumen ModelNet40 31

Índice de Códigos

5.1 Pseudocódigo algoritmo Box Counting 34
5.2 Pseudocódigo algoritmo Box Counting con voxelización 35
5.3 Pseudocódigo algoritmo KNN 38

1 Introducción

El reconocimiento de objetos en 3D es uno de los campos de investigación con mayor crecimiento a lo largo de los últimos años dentro de la inteligencia artificial. Además, no cabe duda de que a lo largo de los próximos años seguirá siendo un área importante pues se aplica en sectores tan dispares como la robótica industrial o doméstica, la conducción autónoma o la biometría, todos ellos en auge.

Como se ha explicado en el preámbulo, en este trabajo se pretende evaluar la eficacia de un nuevo descriptor, un concepto que se ampliará en la Sección 2.3.3, pero que por el momento definiremos como un método que nos aporta información para reconocer objetos sobre las nubes de puntos. Contrariamente a lo que se podría pensar, es frecuente que muchos de los algoritmos del estado del arte en reconocimiento de objetos usando nubes de puntos consistan en transformar la nube a dos dimensiones, proceso conocido como proyección tridimensional, y posteriormente usar algún tipo de clasificador (frecuentemente involucrando redes neuronales convolucionales) sobre estas imágenes Dos Dimensiones (2D). Si bien este método ha probado una gran eficacia, parece intuitivo el hecho de que se están perdiendo características de la nube que se podrían explotar con técnicas de reconocimiento 3D puro, es decir, aquellas que obvian la fase de convertir a imagen plana la nube de puntos y extraen las características directamente desde los puntos de la nube.

Inicialmente, la alternativa basada en tratamiento de datos puramente 3D daba peores resultados que aquella consistente en proyectar la figura y aplicarle redes convolucionales. No obstante, recientemente se está explorando la vía de trabajar únicamente en 3D con buenos resultados, como se verá ejemplificado en el Capítulo 2.4. Es esta opción, la de trabajar con la dimensionalidad original de los datos la que se utilizará en este Trabajo Final de Grado (TFG) para enfrentar el problema del reconocimiento de objetos.

1.1 Dimensión fractal

En el Capítulo 2 se explicará más en profundidad qué son los fractales y qué es la dimensión fractal. No obstante, por ahora diremos que es una propiedad que presentan los objetos que nos da una medida de cómo cambia su tamaño en función de la precisión con la que este se mide. Esta propiedad está relacionada con la complejidad de las figuras (B. B. Mandelbrot, 1983), es por ello que en áreas del conocimiento como el análisis de señales haya sido usada para cuantificar como de caóticas o complejas son.

Existen múltiples formas de calcularla, algunas de las cuales se explicarán en el Capítulo 2, pero en este trabajo se utilizará la conocida como dimensión de *box counting* (o conteo de cajas, en castellano). Pese a que cada uno de estos métodos pueden dar resultados distintos para un mismo objeto, todos tratan de cuantificar la misma propiedad.

2 Marco Teórico

En este capítulo se presentarán a nivel teórico los conceptos con los que se trabajará a lo largo de todo el proyecto. Por una parte, se introducirán tanto las nociones matemáticas necesarias para entender los fractales, como las relacionadas con los clasificadores utilizados en Machine Learning.

2.1 Fractales

Los fractales son objetos geométricos investigados en los años 70 por el matemático polaco Benoît Mandelbrot y definidos en su libro *La geometría fractal de la naturaleza* (B. B. Mandelbrot, 1983). La principal característica que define a estas figuras es la autosimilitud (o autosemejanza) matemática. En función de la intensidad de esta propiedad, podemos distinguir los fractales por tres niveles de autosimilitud:

Autosimilitud exacta: Es aquella dónde las partes (o un conjunto de ellas) son idénticas al todo. Además algunas de estas figuras cumplen la propiedad de invarianza a escala como es la Curva de Koch (Koch, 1904).

Autosimilitud aproximada: También llamada cuasiautosimilitud, aparece en aquellos objetos donde sus partes, o conjunto de ellas, son muy similares (aunque no idénticas) al todo. Aparece en la naturaleza, aunque generalmente, sólo en determinadas partes de objetos. También se puede generar artificialmente añadiendo ruido o algún factor de aleatoriedad a las formulas matemáticas que generan las figuras con autosimilitud exacta. Un ejemplo popular que podemos ver en la Figura 2.1d es el brócoli romanesco.

Autosimilitud estadística: Es el tipo más débil de autosimilitud. En este caso, no se exige que se conserve la apariencia física del todo en las partes o viceversa, sino que se conserven las propiedades estadísticas. Es el tipo de autosimilitud que se presenta con mayor frecuencia en la naturaleza, en elementos como las costas marítimas o las cadenas montañosas.

A partir de la definición de Mandelbrot se desarrollan numerosos ejemplos, algunos de los cuales ya fueron planteados con mucha anterioridad a que se acuñara el término "fractal". Estos ejemplos pueden ser divididos en dos conjuntos, los fractales naturales y los fractales puros o ideales.

Fractales naturales: Como se puede inferir de su nombre, son aquellos que se encuentran de forma intrínseca en la naturaleza. En la Figura 2.1 podemos ver algunos de los ejemplos más típicos.

(a) Célula de Purkinje (tipo de neurona).

(b) Alveolos pulmonares.

(c) Marca de erosión.

(d) Brócoli Romanesco.

Figura 2.1: Algunos ejemplos de fractales naturales muy conocidos (fuente: https://commons.wikimedia.org/wiki/Category:Fractals_in_nature).

Fractales puros: Son aquellos que no se pueden encontrar en la naturaleza, pues son construcciones matemáticas definidas por regla general a través de una fórmula iterativa o recursiva. En la Figura 2.2 podemos ver algunos ejemplos clásicos de estos fractales. Como se puede apreciar en esta figura, los fractales ideales no tienen por qué existir sólo en 2 dimensiones, de hecho, en este trabajo nos centraremos en fractales tridimensionales.

2.1.1 Dimensión fractal

En geometría fractal existe el concepto de dimensión fractal, que no es más que una generalización del concepto de dimensión en geometría clásica (más concretamente, para aquellos objetos que no admiten espacio tangente). Para simplificar, se suele definir como una medida de cuánto aparenta llenar el espacio un objeto conforme se aumenta o disminuye la escala de este. Un claro ejemplo es el copo de nieve de Koch. Esta figura tiene una dimensión topológica de valor 1, sin embargo, no puede ser tratada como una curva, pues conforme nos acercamos a la figura la dimensión entre cualesquiera de sus puntos tiende a infinito. Este fenómeno se conoce con el nombre de Paradoja de la Linea de Costa (B. Mandelbrot, 1993).

Existen múltiples formas de calcular esta propiedad y como se ha adelantado en el Capítulo

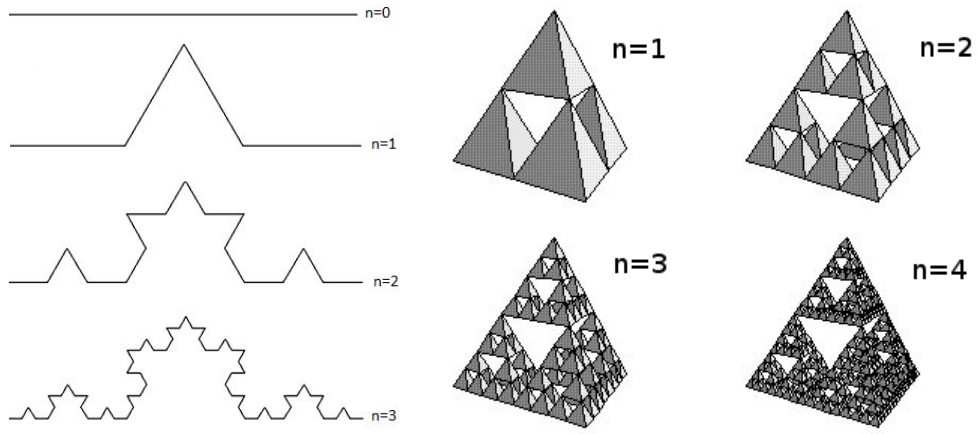


Figura 2.2: Dos populares fractales puros: La curva de Koch a la izquierda (Bonev y cols., s.f.) y el tetraedro de Sierpinski a la derecha (Thäle y Freiberg, 2008).

1 no todos tienen por qué dar idéntico resultado. Sin embargo, todos cuantifican la misma propiedad, la dimensión fractal. A continuación, se introducen muy brevemente algunos de estos algoritmos:

Dimensión de información: Nos permite medir la dimensión fractal de una distribución probabilística, observando la relación entre el crecimiento de la entropía de Shannon (Shannon, 1948) y la discretización del espacio.

Dimensión de correlación: Esta medida de dimensionalidad de conjuntos de puntos es útil por su velocidad de cálculo y precisión. Dado un conjunto de N puntos en un espacio m-dimensional, se calcula el promedio de vecinos de cada punto para cierto radio de vecindad que llamaremos ε. A continuación se variará ε y se repetirá el proceso, creando la función $C(\varepsilon)$ que representa la relación entre número de puntos vecinos y radio de vecindad. Típicamente, esta función toma una forma exponencial dónde la base es ε y el exponente la dimensión de correlación (d_{Corr}), como vemos en la Ecuación 2.1.

$$C(\varepsilon) \sim \varepsilon^{d_{Corr}} \tag{2.1}$$

Una forma interesante de mejorar la eficiencia de este algoritmo es elegir el número de puntos que se usan para promediar el número de vecinos. Esto junto a la potencial paralelización de este calculo, al ser cada punto con su vecindad independiente lo convierten en un método con excelente eficiencia.

Dimensión de Higuchi: Propuesta en 1988 (Higuchi, 1988) como una medida de complejidad de ondas. El proceso consiste en dividir la señal en secuencias temporales, medir la longitud de estas y posteriormente calcular la media de longitudes entre las secuencias. Por último, la dimensión de Higuchi se calcula como la pendiente del ajuste a una

función lineal usando mínimos cuadrados sobre la curva descrita por la Ecuación 2.2.

$$D_{Hi} = \frac{ln(L(k))}{ln(\frac{1}{k})} \tag{2.2}$$

siendo k cada intervalo temporal y $L(k)$ la longitud de ese intervalo.

Exponente de incertidumbre: Método para calcular la dimensión fractal en un sistema de dispersión caótica donde coexisten múltiples atractores. Este método usa trayectorias aleatorias en este sistema que son perturbadas en una pequeña cantidad denotada por ε. La función $f(\varepsilon)$ está definida por el número de trayectorias que cambian de cuenca de atracción dividido entre el número total de trayectorias alteradas. Típicamente esta función será exponencial con base ε siendo el exponente el coeficiente de incertidumbre que se corresponde linealmente con la dimensión de conteo de cajas.

Dimensión de Hausdorff: Descrita en la segunda década del siglo XX (Hausdorff, 1918), mucho antes de acuñarse el término fractal, se calcula en fractales observando cómo aumenta o disminuye la longitud de estos en función del número de iteraciones necesarios para generarlos. En un ejemplo clásico como la curva de Koch para cada iteración cada uno de sus segmentos se convierte en 4 de tamaño 1/3 del segmento original, por ello la Dimensión de Hausdorff de la Curva de Koch se calcula como vemos en 2.3

$$N = S^{D_{Ha}} \rightarrow D_{Ha} = \frac{log(N)}{log(S)} \rightarrow D_{Ha} = \frac{log(4)}{log(3)} \approx 1,26 \tag{2.3}$$

siendo el numerador de la fracción siempre el logaritmo del número de segmentos resultantes de cada uno de los existentes en la iteración anterior (N), y el denominador el logaritmo de la inversa del factor de cambio de longitud de cada uno de estos segmentos (S). Para algunos casos, esta dimensión es equivalente a la de conteo de cajas.

Dimensión de Assouad: Definición relacionada con los conceptos topológicos de bolas y recubrimiento, consiste en calcular el número de bolas necesarias para recubrir la figura fractal en función del radio de estas (Robinson, 2010).

Dimensión de conteo de cajas: Probablemente el método más conocido de los de esta lista. Se usa para determinar la dimensión fractal de conjuntos en un espacio euclídeo, por lo que es un método frecuentemente usado cuando se quiere calcular la dimensión de objetos reales. Este es el algoritmo que se ha utilizado para la realización de este TFG, tanto por su sencillez para trasladarlo a espacios tridimensionales, como por ser uno de los más documentados. Aunque en la Sección 5.1 se encuentran más detalles sobre la implementación de este método y el algoritmo detallado con un pseudocódigo en la Sección 2.1.2, se introducirá a nivel teórico su funcionamiento.

2.1.2 Dimensión fractal por conteo de cajas

El nombre de este método es muy descriptivo, pues el primer paso consiste en dividir el espacio en una cuadrícula con cajas (cuadrados) de tamaño de lado ε. A continuación, se

cuenta el número de estas celdas ocupadas por el objeto cuya dimensión se quiere calcular. Este proceso se repite múltiples veces decrementando ε en una cantidad fija a cada paso con lo que se obtiene la función $N(\varepsilon)$ (número de cuadros requerido para cubrir toda la figura en función de la longitud de su lado). Esta función, al igual que ocurre en varios de los métodos mencionados, es de tipo exponencial con la siguiente forma:

$$N(\varepsilon) = \varepsilon^{D_{BC}} \tag{2.4}$$

donde $\varepsilon^{D_{BC}}$ es la dimensión de *box-counting*. Parece inevitable que esta expresión recuerde a las obtenidas con la dimensión de Haussdorf y la de Higuchi, y al igual que con estas se puede operar aplicando logaritmos para convertirla en una función lineal:

$$log(N(\varepsilon)) = D_{BC} \cdot log(\varepsilon) \tag{2.5}$$

Como se puede ver, en la ecuación lineal resultante D_{BC} es la pendiente.

En la práctica lo que haremos para hallarla será teniendo la función con logaritmos a ambos lados de la igualdad aplicar aproximación por mínimos cuadrados a una función lineal y quedarnos con la pendiente de esta.

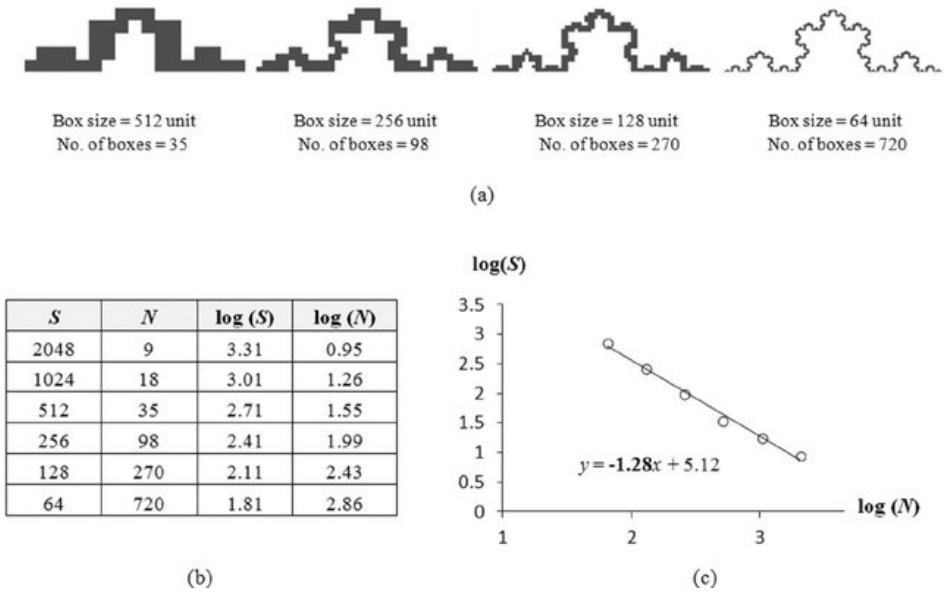


Figura 2.3: Resumen de cómo se calcula la dimensión por *box counting* (Rian y Sassone, 2014). En *(a)* se calcula el número de cajas para cada longitud de lado ($N(\varepsilon)$). En *(b)* vemos una tabla resumen tras aplicar los logaritmos. Por último en *(c)* la recta resultante, cuya pendiente es la dimensión fractal. Usualmente se representa la inversa del número de cajas en el eje X para que la pendiente sea positiva, esto no se aplica en este caso.

Como se puede apreciar en la Figura 2.3, la dimensión obtenida es muy similar a la que habíamos obtenido aplicando el método de Haussdorf (1,28 frente a 1,26). Se establece la

relación entre estos dos métodos de cálculo tan frecuentes como:

$$D_{UBC} = \overline{\lim} \frac{log(N(\varepsilon))}{log(\frac{1}{\varepsilon})} \; ; \; D_{LBC} = \underline{\lim} \frac{log(N(\varepsilon))}{log(\frac{1}{\varepsilon})} \; \Rightarrow \; D_{Ha} \leqslant D_{UBC} \leqslant D_{LBC} \tag{2.6}$$

D_{UBC} y D_{LBC} son lo que se conoce como *upper* y *lower box dimension* (dimensión superior e inferior de conteo de cajas). Estos valores se definen como los límites superior e inferior de la sucesión cuyo límite cuando ε tiende a 0, como ya hemos visto, es la dimensión de conteo de cajas.

Como ya se ha comentado, trasladar este algoritmo a un espacio euclídeo tridimensional es un proceso trivial, pues lo único que se debe hacer es sustituir los cuadrados de la malla por cubos (conformando lo que se conoce como un *voxel grid*) sin alterar ningún otro paso del proceso.

2.2 Clasificadores

En este trabajo se enfrenta un problema de clasificación, es decir, dada una serie de instancias (figuras 3D en nuestro caso), y una lista de categorías a las que pertenecen las instancias se debe asignar una y solo una categoría a cada una de ellas. El objetivo del problema es maximizar el número de instancias clasificadas correctamente de acuerdo a su categoría real. Para asignar las categorías se usa lo que conocemos como clasificadores, que son algoritmos que realizan esta función. Estos algoritmos, como su propio nombre indica, sólo se dedican a asignar categorías en base a la representación vectorizada de cada instancia que reciben, pudiendo ser esta vectores de datos "en crudo", es decir, directamente extraídos de las instancias, o características sintetizadas de ellas. En el caso de este trabajo, las entradas de los clasificadores serán la dimensión fractal, de forma que esta propiedad será usada por los algoritmos para etiquetar cada figura. A continuación, se explicarán los clasificadores con los que se ha experimentado.

2.2.1 K-Vecinos más cercanos

KNN, también conocido como K-vecinos más cercanos en castellano, es un algoritmo de clasificación supervisada en base a la distancia a conjuntos previamente clasificados. Se basa en computar para cada elemento del conjunto de test la distancia de éste con todos los del conjunto de entrenamiento, y cataloga en función de la clase mayoritaria de los elementos más cercanos.

Comúnmente, la diferencia entre instancias se establece como la distancia euclídea entre sus vectores de características. No obstante, existen variaciones que usan otras métricas de distancia.

2.2.1.1 Distancia Euclídea

Es la distancia habitual entre dos puntos del espacio euclídeo que se puede deducir del Teorema de Pitagoras. Sean p y q dos puntos en un espacio euclídeo n-dimensional su distancia

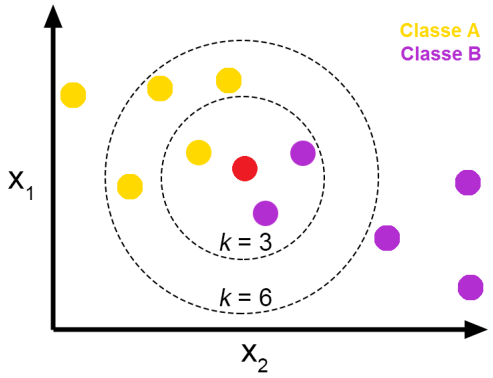


Figura 2.4: Muestra de cómo en función de la K elegida puede variar la clasificación resultante (Suratkar, 2020).

euclídea se calcula como:

$$d_E(p,q) = \sqrt{\sum_{1}^{n} (p_i - q_i)^2} \tag{2.7}$$

2.2.1.2 Distancia de Mahalanobis

Esta es otra medida de distancia menos conocida pero frecuentemente usada en estadística. Se diferencia de la distancia euclídea en el hecho de que tiene en cuenta la correlación entre las variables. El ejemplo clásico para explicar la utilidad de esta distancia es el de un pescador que quiere medir la similitud entre sus peces. Dada la forma alargada de los peces, si para identificar a cada uno de ellos se usa su longitud y su anchura y se mide la diferencia con la distancia euclídea (con la formula de la Ecuación 2.7) la variable longitud tendrá mucho mayor peso sobre el resultado, pues la diferencia de longitud entre dos peces puede ser mucho mayor a la de anchura. Para corregir esto, la distancia de Mahalanobis incluye la matriz de covarianza en la ecuación, lo cual pondera con mayores pesos a las variables con menor desviación típica, haciendo que todas las variables cobren idéntica importancia en la medida de distancia. De esta forma, la fórmula de la distancia de Mahalanobis queda así:

$$d(\overrightarrow{p}, \overrightarrow{q}) = \sqrt{(\overrightarrow{p} - \overrightarrow{q})^T S^{-1} (\overrightarrow{p} - \overrightarrow{q})}$$

siendo S la matriz de covarianza. No obstante, la versión usada definitivamente en este proyecto no es exactamente esta, pues se detectó que no había correlación entre los elementos de los vectores. Para esos casos existe la variación conocida como distancia euclídea estandarizada que se describe a través de la siguiente ecuación:

$$d_M(p,q) = \sqrt{\sum_{1}^{n} \frac{(p_i - q_i)^2}{s_i^2}}$$

siendo s el vector de desviaciones estándar de los elementos de p y q. Esta medida nos posibilita comparar instancias teniendo todas sus propiedades idéntico peso en la comparación, más

adelante, en la Sección 6.2.2.3 se explicará por qué y cómo se usa esta distancia en este proyecto .

2.2.2 Support Vector Machines

Support Vector Machines (SVM) es uno de los algoritmos de clasificación más populares hoy en día para espacios de decisión de gran dimensionalidad. Se basa en separar el espacio de muestras en dos subespacios a través de un hiperplano. Además, SVM siempre obtiene de todos los hiperplanos posibles para separar los datos, el que deja un mayor margen entre las categorías, cuyos puntos más cercanos al hiperplano se denominan vectores de soporte, y el propio hiperplano. La implementación real de este algoritmo suele incorporar un parámetro C de tolerancia, conocido como *soft margin*, que permite clasificar erróneamente algunos elementos a cambio de una penalización. A cambio, permite combatir el *overfitting* que se produce cuando se obtiene el hiperplano de manera estricta. Otro parámetro importante es el de tolerancia, que define criterio de parada que detiene el algoritmo cuando se encuentra un hiperplano que cumple este valor.

Es habitual que no se quiera clasificar sólo en dos clases. Para ello existen múltiples enfoques, pero el usado en este trabajo es el llamado uno-vs-uno (Knerr y cols., 1990). Esta estrategia conlleva construir $numeroClases \cdot (numeroClases - 1)/2$ clasificadores. Si por ejemplo tuviéramos 3 clases (A, B y C), un clasificador se encargaría de discriminar entre A y B, otro entre A y C y el último entre B y C, escogiendo la clase determinada por la mayoría de ellos.

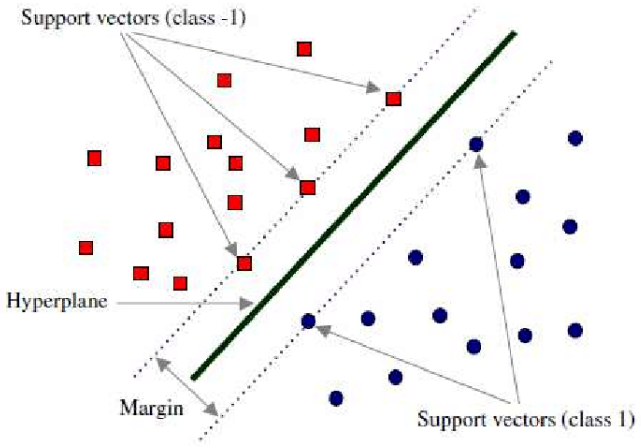


Figura 2.5: Ejemplo de cómo funciona un clasificador SVM en dos dimensiones (Ulaş, 2013).

También se utiliza SVM para resolver problemas de regresión adaptando el algoritmo original. No obstante, no se profundizará ya que el problema enfrentado es de clasificación.

2.2.3 Redes neuronales

Las redes neuronales son unos modelos computacionales cuyos orígenes teóricos datan de la década de los 40's (McCulloch y Pitts, 1943). Pese a la antigüedad de la base teórica, la popularidad de estos algoritmos ha ido en aumento durante las últimas dos décadas, ligada

principalmente al aumento de las capacidades computacionales del *hardware* en el mercado. Una red neuronal básica no es más que un conjunto de nodos de cómputo (neuronas como las que propusieron McCulloch y Pitts) donde cada nodo tiene una serie de entradas y salidas, y estos se organizan por capas de forma que las salidas de la capa N son las entradas de la capa N+1. Una neurona clásica no es más que una función aplicada a un dato de entrada, y es la elección de esta función y de sus parámetros lo que varía el comportamiento de cada una de ellas. Además, cada una de las entradas a las neuronas está ponderada por un peso y para cada capa de entradas hay también un peso fijo llamado sesgo (o *bias* como se conoce normalmente en inglés).

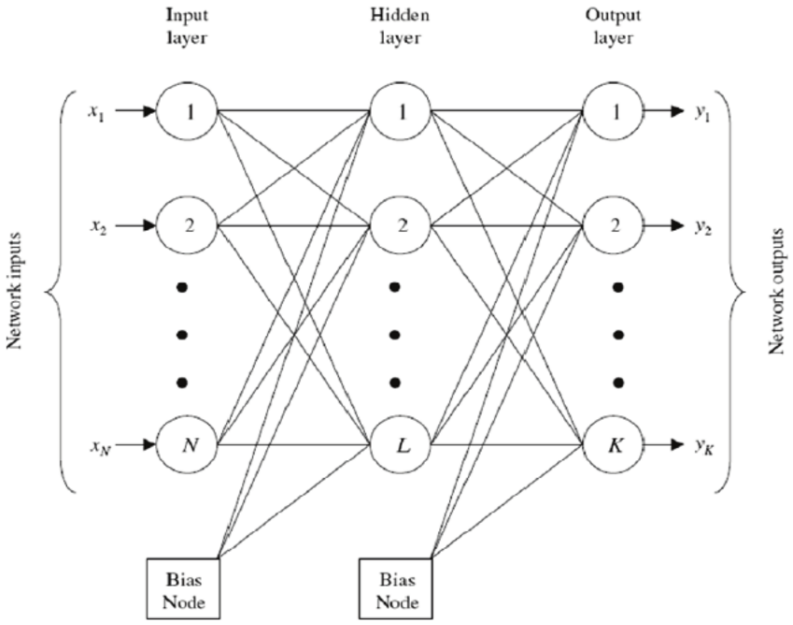


Figura 2.6: Esquema de una red neuronal clásica (Lahiri y Ghanta, 2009).

En la Figura 2.6 podemos ver cómo las neuronas se organizan en capas y cómo en cada capa hay un bias densamente conectado con la capa siguiente. Además, este esquema introduce el concepto de capas ocultas, que son todas aquellas capas entre la de entrada y la de salida, que resultan invisibles a todo aquel que utiliza el clasificador como una caja negra. El entrenamiento de una red neuronal consiste en regular los pesos de la red para perfeccionar la salida de esta, y uno de los métodos más populares para lograr esto es el algoritmo de descenso por gradiente, que utiliza *backpropagation* (o propagación hacia atrás) para calcular el incremento del error. Este método consiste en introducir una serie de entradas cuya salida esperada conocemos y calcular la diferencia entre el resultado esperado y el real (el error en la red). Una vez se conoce este error, se propaga hacia atrás (desde la salida de la red hacia la entrada), de forma que se van corrigiendo los pesos para minimizar el error en la salida. Este proceso se realiza de forma iterativa, mejorando poco a poco la precisión del modelo. Como se ha explicado antes, lo que caracteriza a cada neurona es su función de activación. Hoy en día existen multitud de éstas siendo algunas de las más conocidas la función sigmoidea, la tangente hiperbólica, ReLu o la softmax (exponencial normalizada) entre otras.

2.2.3.1 Perceptrón multicapa

El perceptrón es considerado la primera neurona artificial de la historia (Rosenblatt, 1958). Al igual que las neuronas artificiales ya explicadas, el perceptrón es una unidad de cómputo que dadas unas entradas les aplica una función obteniendo sus salidas correspondientes. La única particularidad que presenta es su función de activación, que es la conocida como función escalón (o *step* en inglés), descrita como vemos en la Ecuación 2.8a.

$$f(x) = \begin{cases} 0 & si \quad x < 0 \\ \\ 1 & si \quad x \geq 0 \end{cases} \tag{2.8a}$$

$$f(x) = \frac{e^x}{e^x + 1} \tag{2.8b}$$

$$f(x_i) = \frac{e^{x_i}}{\sum_0^j e^{x_j}} \tag{2.8c}$$

$$f(x) = \max(0, x) \tag{2.8d}$$

Las ecuaciones 2.8 corresponden a las funciones escalón, sigmoidea, ReLU (rectificadora) y *softmax* respectivamente, en la Figura 2.7 vemos representadas las tres primeras. La cuarta, la función *softmax* es usada en problemas de clasificación multiclase como el que aquí enfrentamos para convertir las puntuaciones de cada clase (intervalo $(-\infty, \infty)$) en probabilidades (intervalo $[0, 1]$), por ello se usa en la salida de las redes.

Un perceptrón multicapa consiste simplemente en una red neuronal como las ya explicadas donde todas las neuronas son perceptrones. El mayor inconveniente de los perceptrones es que sólo permiten clasificar datos linealmente separables. El uso de estas redes multicapas que concatenan múltiples perceptrones permite trazar fronteras de decisión más complejas, no obstante, esta limitación siempre existirá mientras las funciones de activación que se usen sean funciones lineares (que es en definitiva lo que caracteriza a los perceptrones frente a otras neuronas). De hecho, esta limitación lleva a que frecuentemente se use el término Multi-Layer Perceptron (perceptrón multicapa) (MLP) para referirse a redes que combinan perceptrones con neuronas de activación no lineal, lo cual sí permite definir fronteras de decisión no lineales, como vemos en la Figura 2.8.

2.2.3.2 Redes neuronales convolucionales

Las Convolutional Neural Network (red neuronal convolucional) (CNN) son redes neuronales a las que se añade las operaciones de convolución y, frecuentemente, la de puesta en común (*pooling*). Son ampliamente usadas en visión artificial, sobretodo en lo que respecta a reconocimiento y segmentación de imágenes.

Las capas de convolución se encargan de aplicar la operación con este mismo nombre sobre los datos de entrada (imágenes). La convolución recorre con una serie de máscaras (comúnmente llamados kernels) la imagen de entrada de forma que se obtiene una matriz de características de esa imagen.

Cada kernel tiene un propósito; por ejemplo, algunos son útiles para aristas, y es la combinación de varios de estos lo que nos permite obtener una matriz de características que describa

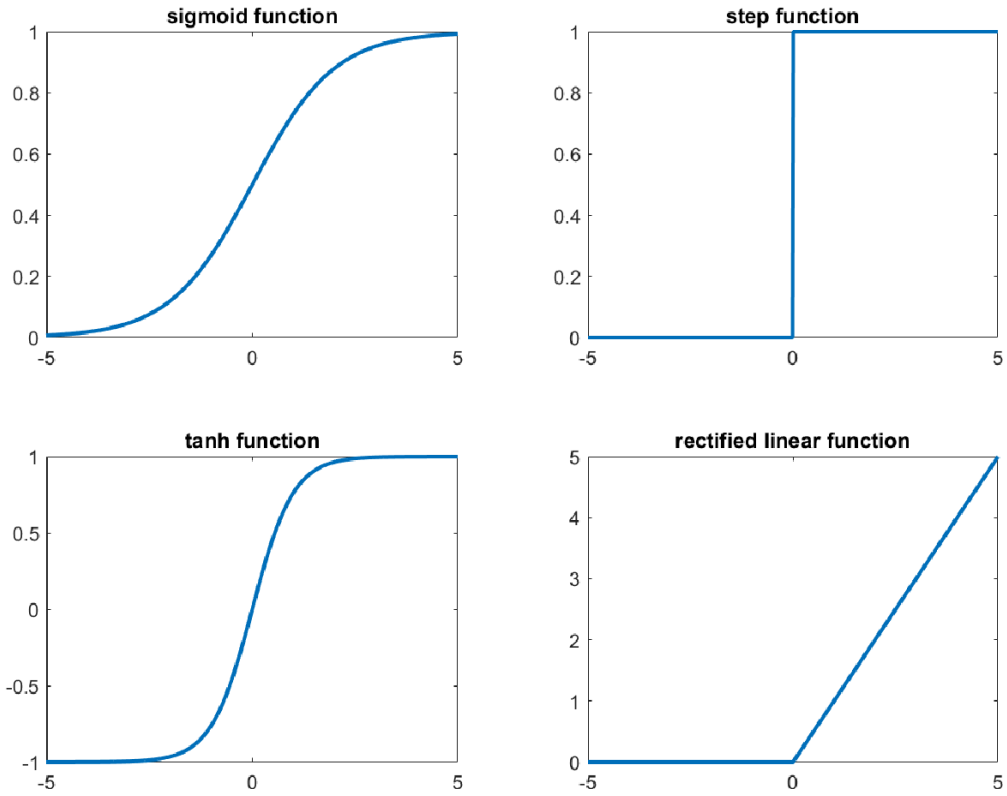
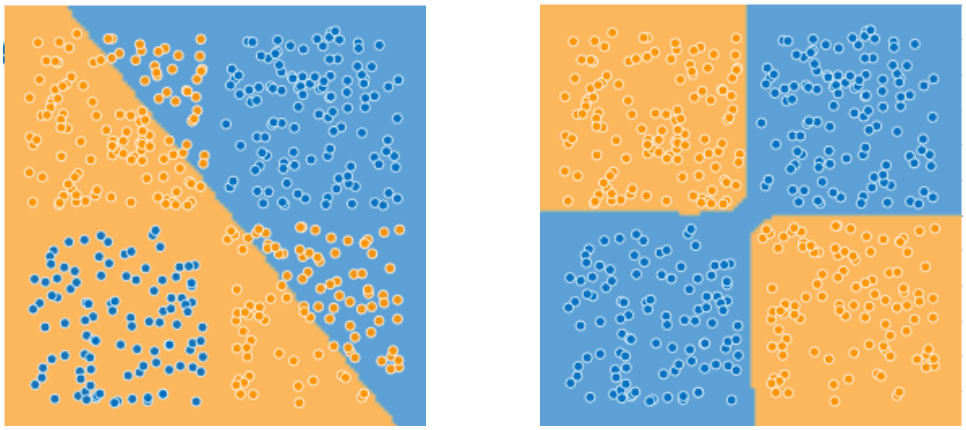


Figura 2.7: Algunas de las funciones de activación más utilizadas (Shao, 2016).



(a) Usando función escalón. **(b)** Usando sigmoidea.

Figura 2.8: Fronteras de decisión encontradas sobre datos no linealmente separables con distintas funciones de activación.

fielmente la imagen para reconocerla. Si, por ejemplo, en nuestra aplicación es crucial detectar aristas verticales, usando filtros con ese propósito simplificamos algo tan complejo como una imagen con color a una descripción del número, posición y longitud de sus aristas verticales. El otro tipo de capas, son las de *pooling* (o puesta en común). Estas realizan un proceso de

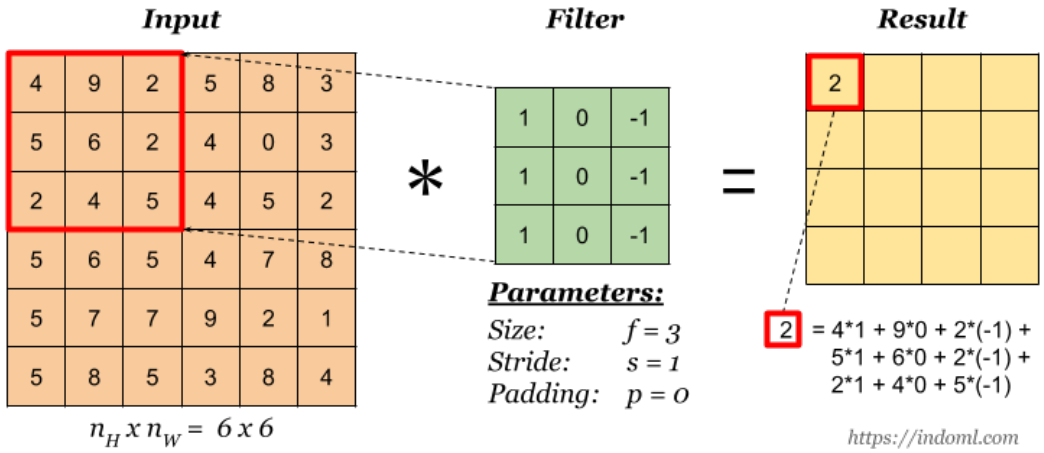


Figura 2.9: Esquema de cómo se calcula el resultado de aplicar cada filtro convolucional (Ng y cols., s.f.).

agrupación y resumen de las matrices obtenidas con la convolución de forma que se reduzca su dimensionalidad, lo cual permite trabajar mejor a las neuronas de la red y reducir el coste computacional. Al igual que la convolución, se realiza aplicando un filtro, como vemos en la Figura 2.10. Como se puede observar, existen múltiples estrategias de *pooling*, como el *max pooling* que se queda con el valor máximo de cada subregión o el *average pooling* que se queda con la media de los valores de ésta.

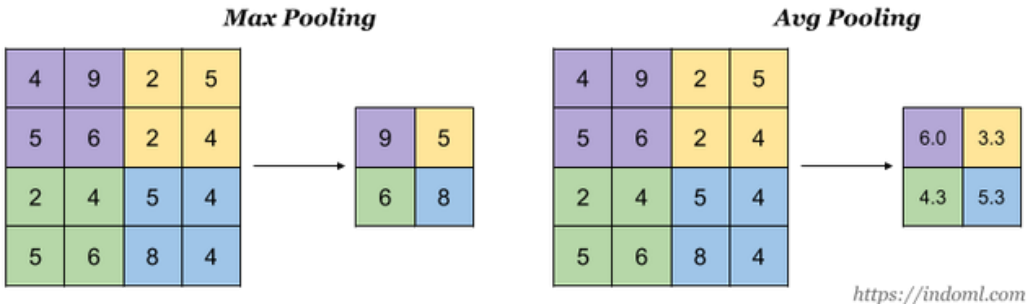


Figura 2.10: Esquema de cómo se calcula el resultado de aplicar la puesta en común sobre una imagen (Ng y cols., s.f.).

2.3 Conceptos de visión artificial

En esta sección se expondrán algunos conceptos básicos de visión artificial que ayudarán a entender cómo encuadraría el algoritmo desarrollado en este TFG en un proyecto completo de visión artificial.

2.3.1 Formas de representar objetos 3D

En este proyecto se trabaja con tres formas de representar objetos 3D que se enumeran a continuación, a lo largo de esta memoria se explicará el porqué de cada una de ellas. Además en la Figura 2.11.

Malla poligonal: Un formato muy frecuente en diseño por computador, donde los objetos se definen por los polígonos que forman sus superficies, que a su vez, están definidos por sus vértices y sus aristas. De los tres métodos de representación vistos en este punto, este es en lineas generales el que ofrece una mayor fidelidad respecto al objeto real.

Nube de puntos: Es el formato más usado en este trabajo, y en el que se obtienen normalmente las representaciones captadas del mundo real a través de los distintos sensores existentes para ese fin. Una figura se define como un conjunto de puntos, definidos mediante de sus coordenadas, que en algunos casos pueden tener otras propiedades como color o intensidad, no obstante estas no se usarán en este TFG. Uno de los inconvenientes que presenta este formato es la dispersión, es decir, al aparecer las superficies representadas por algunos de sus puntos, estas se muestran de forma inconexa.

***Voxel grid*:** Versión con volumen de los conocidos píxeles, es decir, un conjunto de cuboides situados en el espacio euclídeo. A diferencia de las nubes de puntos, aquí no hay dispersión, de hecho, puede ocurrir el efecto contrario y que un *voxel* ocupe una zona del espacio donde realmente no hay figura.

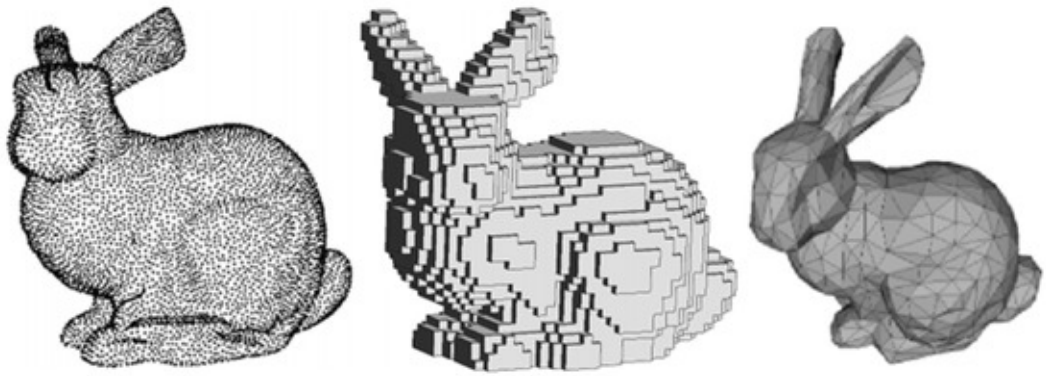
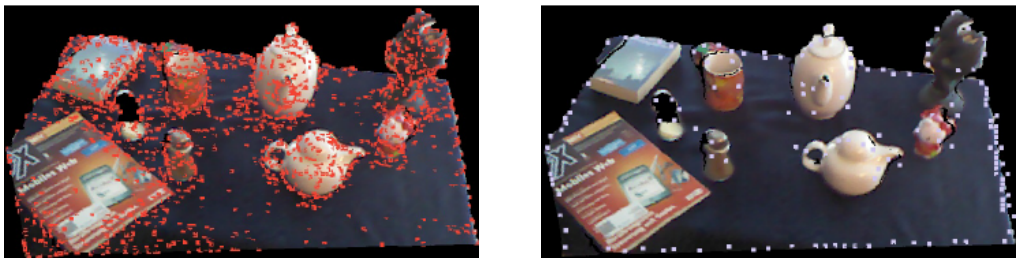


Figura 2.11: Comparativa entre nubes de puntos, *voxel grids* y mallas poligonales (Hoang y cols., 2019).

2.3.2 Detectores

Se denomina detectores a los algoritmos encargados de localizar puntos "destacados" (*keypoints*) en un modelo (ya sea tridimensional como una nube de puntos, o en una imagen plana). ¿Qué puntos se consideran destacados y cuáles no? Eso depende del detector escogido. Algunos se centran en localizar esquinas y otros buscan encontrar las superficies curvas, pero existen infinidad de métodos y cada uno persigue una serie de objetivos muy concretos.

Dependiendo del problema enfrentado se deberán escoger unos u otros. En la Figura 2.12 vemos la diferencia entre como dos de estos detectores localizan puntos significativos en las nubes.



(a) SIFT 3D **(b)** NARF

Figura 2.12: Comparativa de dos populares detectores 3D (Hänsch y cols., 2014).

2.3.3 Descriptores

Los descriptores son algoritmos que calculan, una serie de características (*features*) sobre determinadas regiones de una figura. Existen descriptores locales, que se usan para caracterizar la región de la figura sobre la que se calculan y descriptores globales (como el implementado en este TFG) que se usan para definir toda la figura en base a los valores calculados sobre sus regiones. Un ejemplo básico de descriptor sería un método que dada una nube de puntos nos dijera en qué regiones de esta existen superficies curvas. En la Figura 2.14 se puede ver el resultado de aplicar el descriptor Radius-based Surface Descriptor (RSD) (Marton y cols., 2010) sobre una figura. Normalmente los descriptores se calculan sobre los puntos generados por los detectores de forma que se reduce el coste computacional y se centra la aplicación del algoritmo en zonas que poseen rasgos interesantes de cuantificar. Por este motivo, descriptores y detectores se deben escoger en consonancia, pues determinadas combinaciones funcionan mejor que otras, en la Figura 2.13 vemos el acierto de algunas de estas combinaciones.

Keypoint	Descriptor	Small radius	Large radius
ISS	IntensitySpin	0.99	0.99
ISS	SHOTColor	0.94	0.94
ISS	SHOT	0.93	0.94
ISS	RIFT	0.88	0.71
ISS	ShapeContext	0.82	0.8
Susan	SHOTColor	0.76	0.77
Susan	SHOT	0.76	0.78
Susan	ShapeContext	0.59	0.62

Figura 2.13: Comparativa de acierto entre algunas parejas de detectores (*Keypoint*) y descriptores (*Descriptor*) populares (Bebis y cols., 2016).

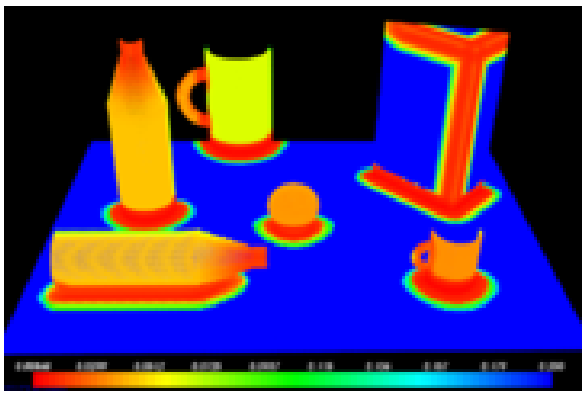


Figura 2.14: Ejemplo del descriptor RSD, cuyo valores máximos se dan en las superficies planas de las nubes de puntos (zonas azuladas en la imagen). Imagen extraída de la presentación de (Marton y cols., 2011)

Es en este apartado donde se enmarca este TFG, en usar la dimensión fractal como descriptor en nubes de puntos 3D para caracterizarlas.

2.4 Trabajos previos y estado del arte

En cuanto a reconocimiento de imágenes, dado el gran potencial de estas técnicas existen multitud de algoritmos muy competitivos y entre los cuales sería difícil seleccionar el mejor, ya que cada uno funciona mejor frente a uno u otro problema y dependiendo de las circunstancias concretas del mismo. Por ello, he querido centrarme en algoritmos que trabajan con el mismo *dataset* que este proyecto, ModelNet, que, como se explica en la sección 4.2, nos presenta en su página web las técnicas que mejores resultados han conseguido clasificando sus figuras.

2.4.1 Reconocimiento 3D proyectando a 2D

A fecha de hoy (Abril 2020) el líder de la clasificación para ModelNet40 es indiscutiblemente RotationNet (Kanezaki y cols., 2016). Este algoritmo se enmarca dentro de la clasificación expuesta en el Capítulo 1 entre aquellos que calculan una serie de transformaciones del objeto 3D y usan estas proyecciones en 2D resultantes como entrada a una red neuronal convolucional. La forma de calcular proyecciones que usa este modelo sobre la figura tridimensional consiste en captar imágenes de esta desde distintos puntos de vista. El nombre de RotationNet viene del hecho de que la cámara toma varias fotografías rotando alrededor de la figura. Además, su algoritmo permite trabajar sólo con algunas de estas imágenes; es decir, con un subconjunto representativo de los puntos de vista alrededor del modelo, lo cual permite una mejor extrapolación al mundo real, donde es frecuente que no se puedan capturar planos de los objetos desde muchos ángulos diferentes. El proceso de entrenamiento con el que se consigue esto lo encontramos resumido en la Figura 2.15.

Adicionalmente, su sistema es capaz de inferir la pose desde la que se ha tomado cada imagen. Dada una serie de imágenes de una silla tomadas desde distintos ángulos sabría que el objeto es una silla y además desde qué perspectiva se ha tomado cada una de las capturas. Para la clasificación, prueban tres arquitecturas densas convolucionales de probada solvencia

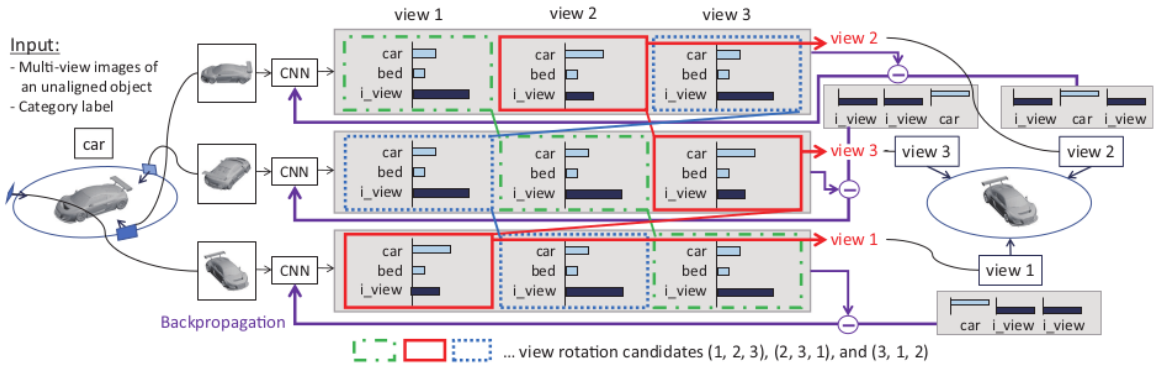


Figura 2.15: En esta imagen se resume el proceso de entrenamiento de RotationNet. En este ejemplo se tiene una instancia de entrenamiento que consiste en tres imágenes y la clase a la que pertenece su modelo. Este esquema simplifica a 2 las categorías entre las que elegir, *car* y *bed*. La tercera, *i_view* representa *incorrect view*, y se usa como medida del posible error. Una red CNN calcula un histograma entre estas tres categorías para cada una de las fotografías. Para cada imagen, se elige a qué punto de vista corresponde en función de su histograma, y viendo qué la clase más representada en las tres vistas es *car* se propaga (con *backpropagation*) esta información a todos los parámetros de la red CNN para su optimización. Imagen extraída directamente del artículo (Kanezaki y cols., 2016).

que han resultado exitosas en las competiciones de clasificación de imágenes más importantes: AlexNet (Krizhevsky y cols., 2012), ResNet-50 (He y cols., 2016) y VGG-M (Chatfield y cols., 2014). Es con esta última arquitectura, con VGG-M, con la que se consiguen los mejores resultados, que ascienden hasta un 98,46% de precisión en ModelNet10 y un 97,37% de precisión en ModelNet40, superando a día de hoy al resto de algoritmos que encontramos en la web de ModelNet.

Otro ejemplo de algoritmo que realiza operaciones de proyección a dos dimensiones sobre las nubes de puntos se puede encontrar en LonchaNet (Gomez-Donoso y cols., 2017). En este trabajo, el proceso de conversión a 2D consiste en tomar tres secciones de cada modelo 3D, cada una de ellas correspondiente a uno de los tres ejes, algo similar a lo que se hace en dibujo técnico cuando se describe un modelo 3D a través de su planta, alzado y perfil. Para conseguir esto, se calcula el centro de cada eje y se recorta una sección de este con un grosor del 5% del total. A continuación, los puntos contenidos en ese 5% se proyectan sobre imágenes planas de 500 X 500 en blanco y negro (binarias), este proceso lo podemos ver esquematizado en la Figura 2.16.

Las nubes de puntos presentan el problema de la eventual dispersión de algunos de sus puntos ya que al realizar las subdivisiones en el paso anterior una superficie plana podría quedar infrarrepresentada en función de cómo de cercana esta sea al centro del eje sobre el que se realiza el corte. Para solventar este problema, se realiza un proceso de dilatación sobre los puntos, de forma que el área de cada uno se convierte en 10 píxeles. En cuanto al método de clasificación utilizado, en este *paper* se emplea una red neuronal profunda formada por tres redes del tipo GoogLeNet (Szegedy y cols., 2015), cada una encargada de procesar una de las tres vistas de la figura. GoogleNet es una famosa arquitectura de red neuronal convolucional compuesta de 27 capas frecuentemente usada en reconocimiento de imágenes. Estas tres redes

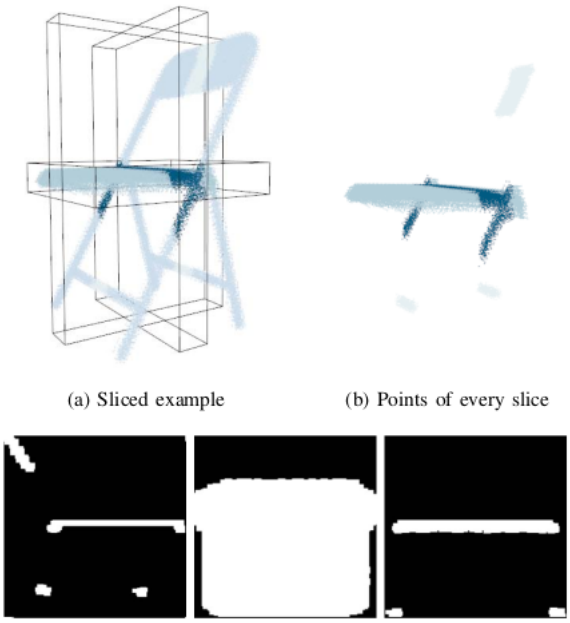


Figura 2.16: Ejemplo de cómo LonchaNet fracciona las figuras con una división por cada eje y cómo estas divisiones se plasman en 2D. Imagen extraída directamente del artículo (Gomez-Donoso y cols., 2017).

GoogleNet se unen en una capa de concatenación para terminar la arquitectura de LonchaNet con una capa densamente conectada. Con el uso de una red propia independiente para cada vista del objeto, lo que se consigue es que estas se especialicen, en la Figura 2.17 encontramos la arquitectura de LonchaNet. Una de las principales ventajas de este tipo de métodos es que frecuentemente requieren menos usos de recursos de cómputo al discretizar la información tridimensional a algo tan reducido como en este caso tres imágenes binarias de 500×500 píxeles.

2.4.2 Reconocimiento 3D sin proyecciones

Por otra parte, tenemos los algoritmos de clasificación que trabajan directamente sin calcular proyecciones previas. En este grupo también encontramos buenos resultados como es el caso de SO-Net (Li y cols., 2018). En dicho artículo hacen uso de lo que se conoce como Self-Organizing Map (SOM). Las SOM son redes neuronales utilizadas habitualmente para crear representaciones discretizadas de las instancias de entrenamiento de manera no supervisada. Utilizan aprendizaje competitivo, como las Generative Adversarial Networks (GAN). El algoritmo que se sigue comienza por inicializar aleatoriamente los pesos cada nodo (hay un nodo por cada punto en del conjunto de entrenamiento). El siguiente paso es escoger aleatoriamente uno de los vectores de pesos y buscar qué nodo tiene los pesos más parecidos a los del vector seleccionado. A continuación, se calcula la vecindad del nodo escogido en el paso anterior y cuanto más cercano es un nodo al escogido más se le premia haciendo que su vector de pesos se parezca al que se había elegido inicialmente. Este proceso se repite tantas iteraciones como se considere necesario, consiguiendo agrupar las distintas instancias de cada

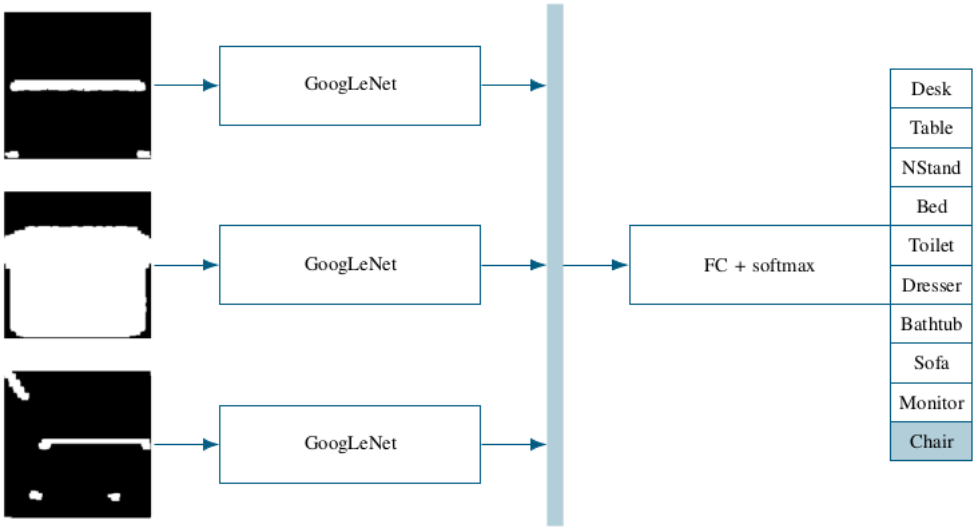


Figura 2.17: Arquitectura de LonchaNet. Imagen extraída directamente del artículo (Gomez-Donoso y cols., 2017).

clase (este tipo de técnicas se conocen como *clustering*) (Ralhan, 2018).

A través de este tipo de redes, SO-Net (Li y cols., 2018) realiza una extracción de características jerárquica. Estas características llegan a un *autoencoder*, otro tipo de red neuronal que aprende la codificación de los conjuntos de entrenamiento ignorando el ruido que puedan tener. Generalmente, estos se componen de dos partes: un codificador, que aprende a convertir las entradas en un formato de datos que, a través de la otra parte, el decodificador, es capaz de reconvertir al formato de entrada. Por último, una serie de capas neuronales densamente conectadas permiten la clasificación. Las principales ventajas que ofrece este método son la invariabilidad a permutaciones y la eficiencia en el entrenamiento.

Siguiendo con otro algoritmo que trata las nubes de puntos directamente encontramos PointNet (Garcia-Garcia y cols., 2016). Lo que se hace en este proyecto es subdividir las figuras con un *voxel grid*; es decir, cierta vecindad de puntos de la nube se ve representada por el *voxel* (cubículo) que ocupa su lugar, y a este *voxel* se le asigna un valor en función de la densidad de puntos que contenga. Para computar esto de forma eficiente lo que se hace es construir la rejilla y escalar la nube de puntos para introducirla en esta de forma que cada punto de la nube queda dentro de un *voxel*, y al ser todos los *voxels* del mismo volumen con contar el número de puntos dentro de con cierto *voxel* conoceremos su densidad. Este proceso lo podemos ver ejemplificado en la Figura 2.18. Para la clasificación de las figuras se usa una red neuronal con operaciones de convolución y *pooling* tridimensionales cuyo dato de entrada es directamente la densidad de puntos de cada *voxel*.

2.4.3 Aplicación de la dimensión fractal

Pese a no ser un concepto muy popular, la dimensión fractal ya ha sido probada como descriptor en ocasiones anteriores, con resultados bastante satisfactorios. En el ámbito del reconocimiento de plantas el trabajo, *Fractal dimension applied to plant identification* (Bruno y cols., 2008) investigó dos algoritmos de cálculo de dimensión fractal, uno de ellos el de conteo

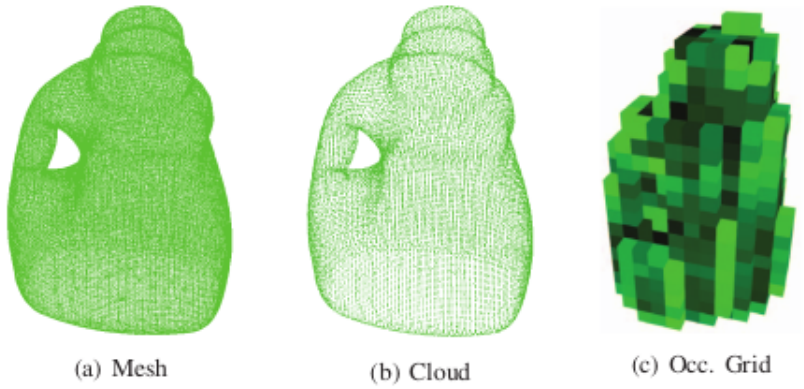


Figura 2.18: Ejemplo de cómo PointNet pasa de malla poligonal a nube de puntos y de ahí a *voxel grid* dónde cada *voxel* tiene asignado un conjunto de puntos y una densidad. Imagen extraída directamente del artículo de (Garcia-Garcia y cols., 2016)

de cajas, en el que nos centramos en este trabajo. En ese trabajo se calcula la dimensión fractal para imágenes de dos dimensiones; no obstante, la base teórica es la misma. Concretamente, el artículo se centra en el reconocimiento de hojas de plantas. Para esto, comienzan segmentando las venas de las hojas y los bordes de estas, como vemos en la Figura 2.19, y es sobre esta información segmentada sobre la que se calcula la dimensión fractal. Se consigue un acierto promedio en la clasificación entre cuatro especies de Passiflora del 97.775%.

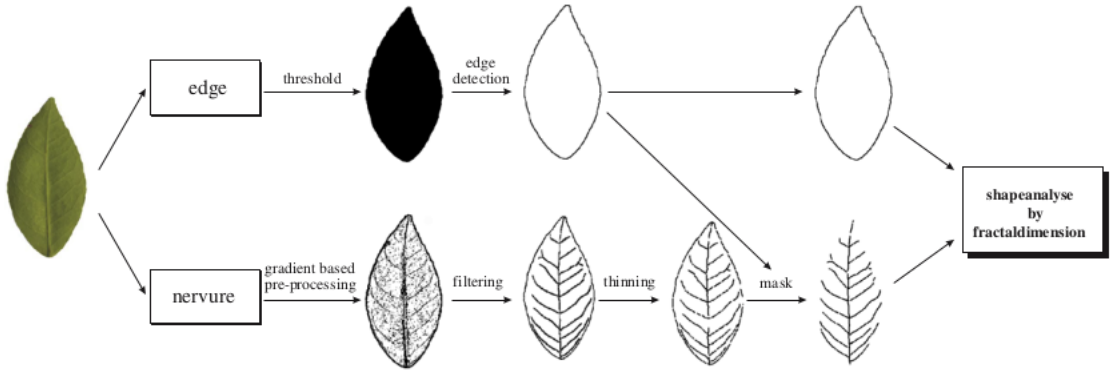
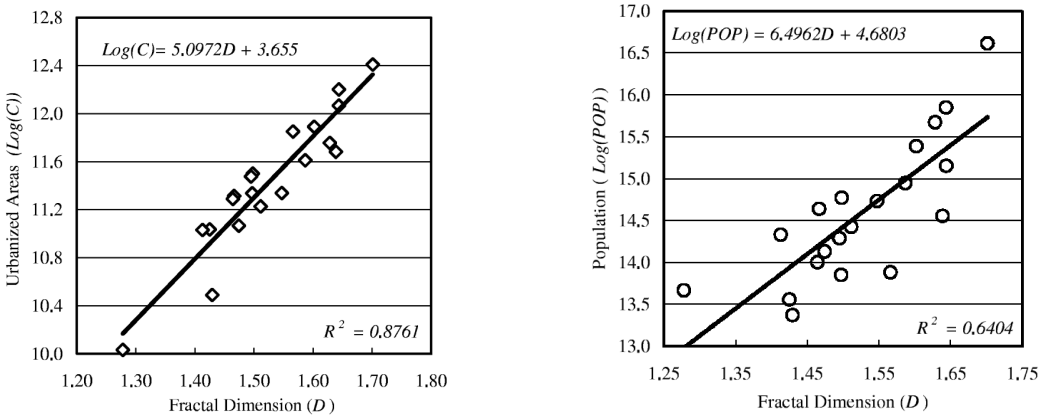


Figura 2.19: Muestra de cómo segmentan las hojas separando el trazado de su contorno de las hojas del de sus venas, se calcula la dimensión por separado sobre cada una de los dos trazados. Imagen extraída directamente del artículo (Bruno y cols., 2008).

Otro ejemplo de uso de la dimensión fractal como descriptor lo encontramos en *Fractal dimension and fractal growth of urbanized areas* (Shen, 2002). En este *paper*, de carácter más teórico, también se usa el método de conteo de cajas para calcular la dimensión fractal, y se pretende demostrar cómo la geometría fractal es capaz de describir propiedades del desarrollo urbanístico que escapan a la geometría euclídea. En este artículo se usan imágenes binarias de mapas de 20 ciudades norteamericanas. En el estudio se halla que todas las ciudades presentan una dimensión en el intervalo $(1, 2)$, y además se prueba la relación entre la dimensión fractal

y el nivel de urbanización de cada ciudad como se puede ver en la Figura 2.20. También se demuestra, que por el contrario no existe relación clara entre la dimensión y la densidad de población como vemos representdo en esa misma figura.



Nivel de urbanización frente a dimensión fractal. Población frente a dimensión fractal.

Figura 2.20: Gráfico que ilustra la relación entre la dimensión de conteo de cajas con el desarrollo urbanístico y la población de cada una de las 20 ciudades analizadas. Imágenes extraídas directamente del artículo de (Shen, 2002)

Un último ejemplo de aplicaciones de la dimensión fractal lo encontramos en el artículo *Use of the Higuchi's fractal dimension for the analysis of MEG recordings from Alzheimer's disease patients* (Gómez y cols., 2009). En él se usa esta propiedad en caracterización de señales, concretamente se aplica sobre las salidas de un Magnetoencefalograma (MEG). La dimensión fractal es considerada un indicador de la complejidad de una señal y para el análisis de señales esta se suele calcular mediante el algoritmo Higuchi (Higuchi, 1988). En dicho trabajo, se pretende comprobar la viabilidad de la dimensión fractal de Higuchi como detector de pacientes de Alzheimer. En este proyecto se logra hasta un 87,8% de acierto (*aciertos/numero instancias*) usando los cinco canales. Además, también usando los cinco canales consigue una especificidad (*negativos bien clasificados / total negativos*) del 95,24%. En la Figura 2.21 se aprecia como cambia el valor de la dimensión entre sujetos sanos y pacientes diagnosticados.

Existe un denominador común entre todos estos métodos del estado del arte y es que más allá del método que usen para calcular la dimensión fractal todos trabajan en dos dimensiones. De hecho, no se han encontrado trabajos previos que utilicen directamente esta propiedad sobre datos 3D para reconocimiento de objetos.

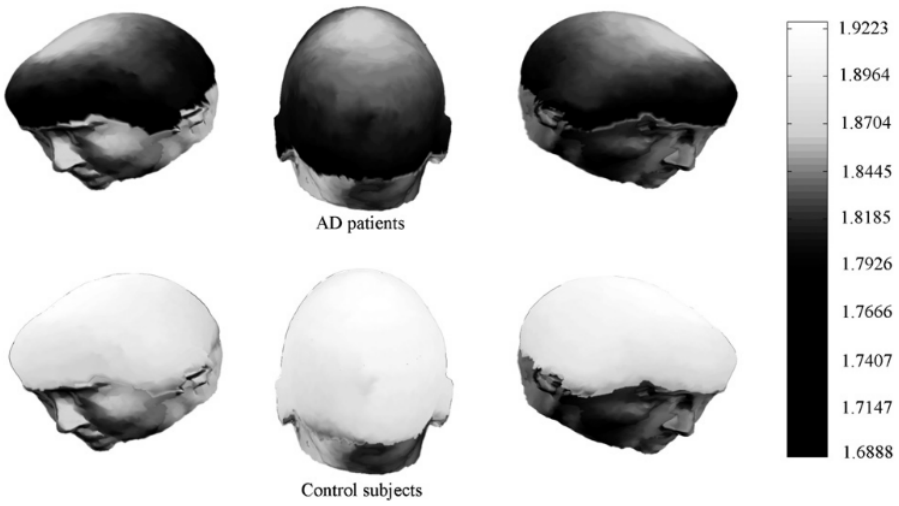


Figura 2.21: Comparativa que muestra una menos dimensión fractal calculada sobre la señal del MEG en pacientes con alzhéimer. Imagen extraída directamente del artículo de (Gómez y cols., 2009)

3 Objetivos y motivación

El principal objetivo de este trabajo es evaluar la viabilidad del uso de la dimensión fractal como descriptor en aplicaciones de reconocimiento de objetos en tres dimensiones.

En cuanto a la motivación, a nivel personal viene dada por el interés en un proyecto que combina un concepto tan sorprendente y a menudo desconocido como la geometría fractal con aplicaciones prácticas en la vida real en un ámbito como la visión artificial. Además, otra gran fuente de motivación es el hecho de que no haya un gran volumen previo de investigación acerca de las aplicaciones de esta propiedad fractal aplicada a reconocimiento de objetos 3D y, por tanto, existe aún un gran margen de progreso.

Más en detalle, los objetivos que se persiguen son:

- Implementar un programa para calcular la dimensión fractal por el método de conteo de cajas sobre modelos en tres dimensiones y secciones de estos. En otros trabajos previos ya se ha trabajado con el cálculo de esta propiedad con otras herramientas; no obstante, en este proyecto se quiere calcular sobre partes del modelo para usarlo como descriptor.

- Comparar la precisión de la dimensión fractal como dato de entrada para distintos tipos de algoritmos de clasificación ampliamente utilizados hoy en día. Dado que este es un campo escasamente investigado, se deberán probar múltiples alternativas para confirmar o desmentir la fiabilidad de usar este descriptor.

- Usar la librería Open3D para el procesado de los modelos. Como se explicará en la sección 4.1, esta es una herramienta relativamente joven y por tanto existen aún pocos ejemplos de uso disponible. Por ello, es interesante desarrollar el proyecto haciendo uso de esta.

- Probar la utilidad de la dimensión fractal como descriptor 3D consiguiendo una medida de precisión a la altura de las presentadas en *ModelNet Benchmark Leaderboard* (se detallará en la Sección 4.2) o demostrar la imposibilidad de usar este método como descriptor si esto no se consiguiera.

4 Metodología

En este capítulo se expondrán las herramientas utilizadas para la realización del proyecto tanto para el procesado de modelos como la clasificación de estos.

4.1 Open3D

Open3D (Zhou y cols., 2018) es la herramienta escogida para el procesado de modelos en tres dimensiones en este proyecto. Es una librería de código abierto compatible tanto con el lenguaje Python (que es el que se usará en este TFG) como con C++. La decisión de usar esta librería se basa en el interés por explorar una herramienta tan joven como esta (las primeras versiones son del 2018 y en abril de 2020 sigue en la versión 0.9.0) y en sus perspectivas de excelente eficiencia y claridad en el código. El principal inconveniente de usar un software tan nuevo es la falta de ejemplos en la red. No obstante, esto se compensa con una documentación bien estructurada y que no deja de mejorar.

4.2 Dataset

El conjunto de datos (lo que comúnmente se conoce como *dataset*) utilizado para evaluar el método propuesto es Princeton ModelNet (Wu y cols., 2015). Este conjunto de datos ha sido elegido tanto por su accesibilidad como por su popularidad entre los investigadores, popularidad debida en gran medida a que las clases que en él aparecen son fruto de un estudio sobre las categorías de objetos más comunes. Además, en la propia página web de ModelNet [1] encontramos una clasificación de los algoritmos que han conseguido mejores resultados hasta la fecha tanto en ModelNet10 como en ModelNet40, que recibe el nombre de *ModelNet Benchmark Leaderboard*.

Este *dataset* se puede utilizar para múltiples problemas, como segmentación o estimación de pose; no obstante, nosotros lo usaremos para resolver un problema de clasificación, es decir, dadas las instancias de *test*, decir a qué categoría pertenece cada una.

Para evaluar el correcto funcionamiento de los algoritmos implementados se usará únicamente una métrica, el acierto o precisión, que definimos como el número de instancias bien clasificadas dividido entre el total de instancias (multiplicado por 100 si queremos expresarlo como porcentaje). Además, también se usará un tipo de gráfico bidimensional conocido como matriz de confusión, que representa en un eje la categoría correcta (eje Y en nuestra implementación) y en el otro la categoría inferida por los algoritmos (eje X). Para cada par de categorías se muestra el número de instancias de la clase del eje Y clasificadas como la clase del eje X entre el total de instancias de la clase del eje Y. Por ejemplo, para la celda situada en $X = laptop$ e $Y = monitor$ de la matriz el valor que aparece será el número de instancias

[1] https://modelnet.cs.princeton.edu/

de la clase *monitor* que han sido clasificadas como clase *laptop* entre el total de instancias de la clase *monitor*. A lo largo de este TFG se verán muchos ejemplos de este gráfico, que nos permite entender los motivos que puede haber tras los errores de clasificación a través de analizar las confusiones más frecuentes.

Para usar este dataset se han tenido que solventar dos problemas. Por una parte, los modelos vienen codificados en formato Object File Format (OFF). Este es un formato que almacena las figuras como mallas poligonales descritas por caracteres ASCII. Este formato ofrece una buena portabilidad por su sencillez. En éĺ simplemente se define la posición de una serie de vértices y posteriormente se indican cuáles de estos vértices componen cada una de las caras de la figura. Para solucionar esto, se recurrió a transformar mediante el método ”sample_points_uniformly” de la clase ”geometry.TriangleMesh” de Open3D, que se explicará en el Capítulo 5. Esto permite trabajar con las figuras como nubes de puntos, un formato habitualmente más sencillo de capturar en el mundo real y de procesar. Además se ha detectado un error en Open3D (versión 0.8.0) por el cual cuando se crea un objeto de tipo VoxelGrid a partir de una malla poligonal se ignoran algunas zonas de la figura, que quedan sin cubrir por dicho VoxelGrid.

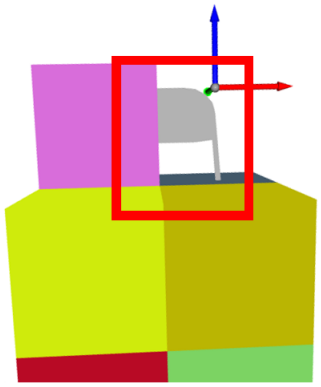


Figura 4.1: Error por el que una parte de la malla (recuadrada) queda sin cubrir por el VoxelGrid.

Por otra parte, el tamaño de las figuras de este *dataset* no está normalizado de forma que el volumen de algunos modelos es hasta dos órdenes de magnitud superior al de otros. Esto no es ningún impedimento para el algoritmo de conteo de cajas a nivel teórico, pues es un proceso invariable a escala; no obstante, a nivel práctico supone un inconveniente. Como se ha explicado en el párrafo anterior, se usa el método ”sample_points_uniformly” para pasar de mallas poligonales a nubes de puntos, y esto se hace siempre con la misma densidad de puntos. Por este motivo, ante diferencias tan grandes en la escala, conseguir la misma densidad en modelos grandes como en pequeños implica inframuestrear las figuras pequeñas y sobremuestrear las figuras grandes. Esto provoca que modelos pequeños queden con una cantidad de puntos no representativa del objeto y los grandes agoten el hardware disponible al tratar de procesar demasiados puntos. La solución a esto fue realizar un escalado a las mallas poligonales con el método ”scale” de la clase ”geometry.TriangleMesh” antes de muestrearlas a nubes de puntos de forma que todas acaben con un volumen similar y por tanto se puedan convertir a nubes de puntos con idéntica densidad de puntos.

Este dataset se divide en dos subconjuntos, ModelNet10 y ModelNet40 en referencia al

número de clases que contiene cada uno.

4.2.1 ModelNet10

Este dataset está compuesto de muebles domésticos, divididos en las siguientes 10 clases:

Clase	Número instancias entrenamiento	Número instancias test
Bañera (bathtub)	106	50
Cama (bed)	515	100
Silla (chair)	889	100
Escritorio (desk)	200	86
Aparador (dresser)	200	86
Monitor (monitor)	465	100
Mesita de noche (night_stand)	200	86
Sofá (sofa)	680	100
Table (mesa)	392	100
Inodoro (toilet)	344	100
Total	**3991**	**908**

Tabla 4.1: Clases de ModelNet10 y cantidad de figuras que las componen.

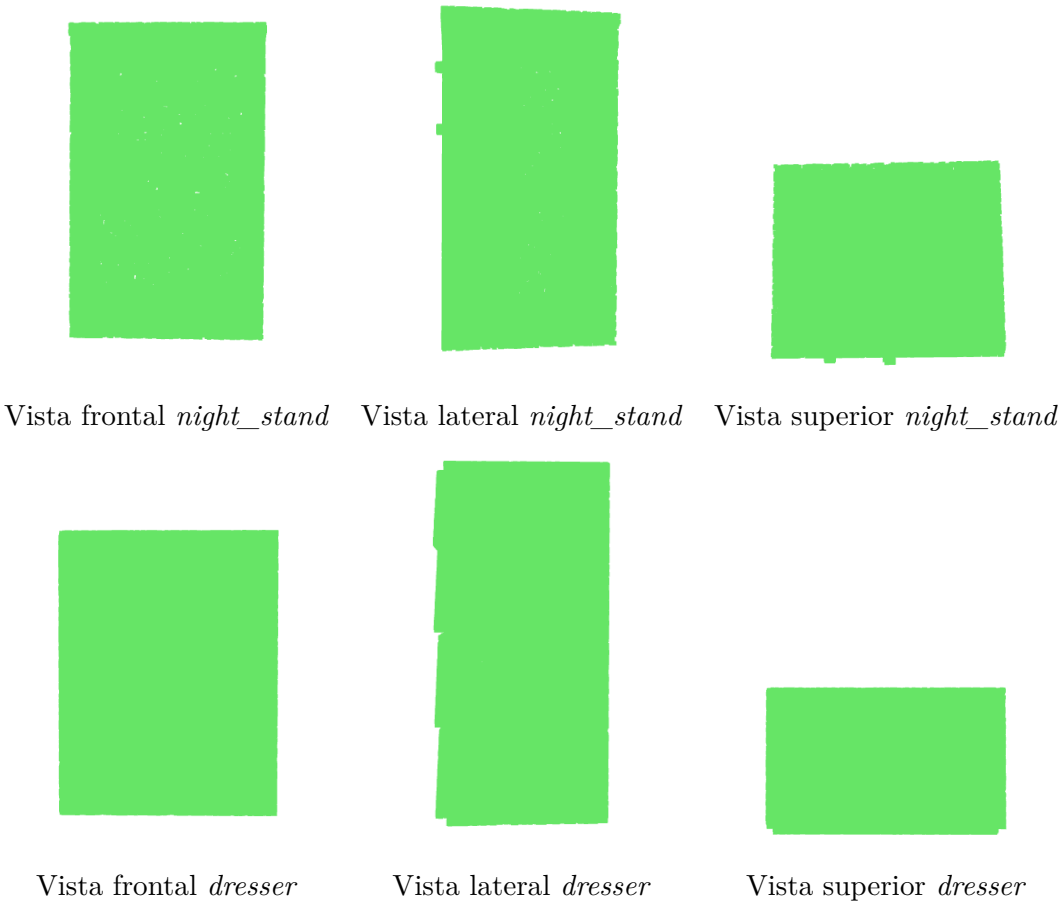
Un problema de este *dataset* es la similitud visual entre algunas de sus clases. Este es un contratiempo que se presenta con mayor relevancia en ModelNet40 como se explicará más adelante, pero en ModelNet10 se da principalmente entre las clases *night_stand* y *dresser*. En la Figura 4.2 podemos ver cómo dos instancias de estas clases se asemejan y resulta difícil discernir a qué categoría pertenece cada una de ellas, esto denota que estamos ante un conflicto semántico: ¿qué diferencia una mesita de noche de un aparador?. Realmente ambos muebles son físicamente muy similares y la diferencia en muchas ocasiones radica en el uso que le da el propietario. Probablemente, los humanos también clasificaríamos erróneamente muchas de estas instancias, no por un error en nuestra visión, sino por el hecho de que ambos conjuntos no son disjuntos y lo que para un observador puede ser una mesita será un aparador para otro. Es evidente que esto no ocurre para todas las instancias de la clase, y hay muchas con diseños claramente más distinguibles que los de la Figura 4.2 que hacen decantar la balanza hacia una de las categorías significativamente. Sin embargo, esta problemática ocurre con suficiente frecuencia para hacer descender sensiblemente la precisión de la clasificación.

4.2.2 ModelNet40

Este dataset añade nuevos muebles a los de ModelNet10, además de otros objetos como vehículos e instrumentos musicales. Este dataset se divide en las 40 categorías mostradas en la Tabla 4.2.

El problema descrito en la Sección 4.2.1 referente al solapamiento de algunas clases se repite en ModelNet40. Además de las ya comentadas colisiones entre las clases *night_stand* y *dresser*, vemos confusiones frecuentes entre, por ejemplo, las clases *door* y *courtain*, algo lógico dado que tanto puertas como cortinas no dejan de ser rectángulos planos más altos

que anchos y poco profundos. Algo similar ocurre con las clases *bench* y *desk*, dado que muchas de sus instancias son una gran base rectangular con cuatro patas. No obstante, el mayor problema en este sentido se presenta entre las clases *flower_pot*, *plant*, *vase* y *cup*. Esta confusión viene dada por el hecho de que, por una parte, en la categoría de macetas aparecen bastantes macetas con plantas en su interior, y que en la categoría de plantas encontramos plantas contenidas en macetas. Por tanto, existen instancias cuya clasificación en una u otra categoría es completamente arbitraria. Por otra parte, la confusión entre jarrón, maceta y copa viene evidentemente dada porque en las tres categorías encontramos figuras similares cuya única diferencias es la escala, el color y el uso que le da el propietario, datos naturalmente no contemplables por algoritmos como el expuesto en este trabajo.



Vista frontal *night_stand* Vista lateral *night_stand* Vista superior *night_stand*

Vista frontal *dresser* Vista lateral *dresser* Vista superior *dresser*

Figura 4.2: Comparativa de las instancias night_stand_0014.off y dresser_0014.off en vista frontal, lateral y superior.

Clase	Número instancias entrenamiento	Número instancias test
Avión (airplane)	626	100
Bañera (bathtub)	106	50
Cama (bed)	515	100
Banco (bench)	173	20
Estantería (bookshelf)	572	100
Botella (bottle)	335	100
Cuenco (bowl)	64	20
Coche (car)	197	100
Silla (chair)	889	100
Cono (cone)	167	20
Copa (cup)	79	20
Cortina (curtain)	138	20
Escritorio (desk)	200	86
Puerta (door)	109	20
Aparador (dresser)	200	86
Maceta de flores (flower_pot)	149	20
Caja de vidrio (glass_box)	171	100
Guitarra (guitar)	155	100
Teclado (keyboard)	145	20
Lampara (lamp)	124	20
Ordenador portátil (laptop)	149	20
Chimenea (mantel)	284	100
Monitor (monitor)	465	100
Mesita de noche (night_stand)	200	86
Persona (person)	88	20
Piano (piano)	231	100
Plant (planta)	240	100
Radio (radio)	104	20
Campana extractora (range_hood)	115	100
Fregadero (sink)	128	20
Sofá (sofa)	680	100
Escaleras (stairs)	124	20
Taburete (stool)	90	20
Table (mesa)	392	100
Tienda de campaña (tent)	163	20
Inodoro (toilet)	344	100
Soporte de TV (tv_stand)	267	100
Jarrón (vase)	475	100
Ropero (wardrobe)	87	20
Xbox	103	20
Total	**9843**	**2468**

Tabla 4.2: Clases de ModelNet40 y cantidad de figuras que las componen.

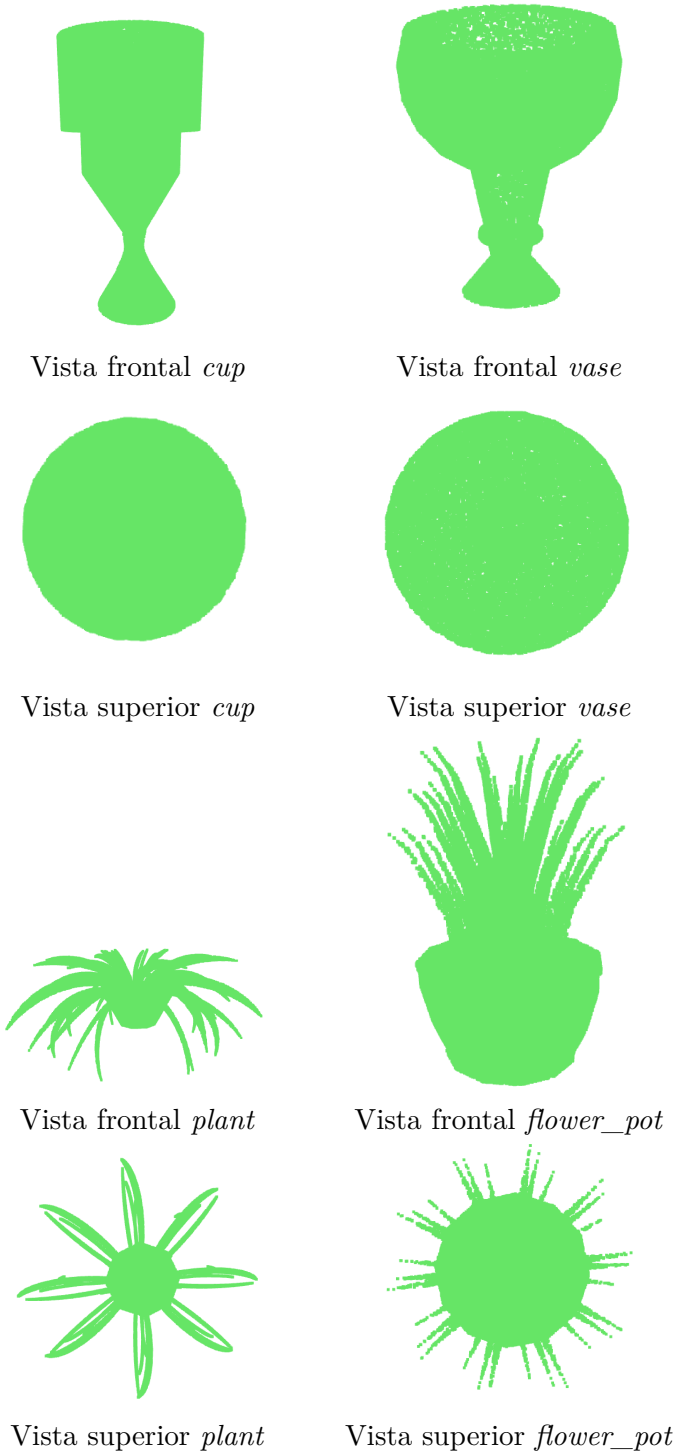


Figura 4.3: Comparativa de las instancias cup_0085.off, plant_0273.off, vase_0528.off y flower_pot_0152.off en vista frontal y superior.

5 Desarrollo

En este capítulo se mostrarán algunos de los algoritmos desarrollados para la realización de este proyecto y se explicarán algunos detalles de su programación. En la primera sección se expondrá cómo se realiza el cálculo de la dimensión fractal, y en las siguientes las implementaciones concretas de los distintos clasificadores cuyos principios teóricos se han introducido en el Capítulo 2.

5.1 Box counting

Este es el programa principal del proyecto. Se encarga de procesar los modelos del *dataset* y calcular los descriptores de cada modelo. El algoritmo tiene como entrada una nube de puntos y como salida proporciona la dimensión fractal de dicha nube y aunque su base teórica ya ha sido explicada en la Sección 2.1.2, aquí se entrará en el detalle de cómo implementar el proceso de cómputo.

El único parámetro a determinar en este método es el número de iteraciones (decrementos del tamaño de caja) que queremos realizar, pues los demás parámetros se calculan en base a este. Se empieza con un tamaño de caja equivalente a la máxima coordenada de la *bounding box* del modelo, de forma que en la primera iteración este siempre quede cubierto por completo con una sola caja. Teniendo este tamaño inicial, el cálculo del decremento a aplicar en cada iteración es trivial, $decremento = tam_0/numeroIteraciones$. A continuación se entra en el bucle principal del método, que crea un VoxelGrid (explicado en la Sección 5.1.4) con *voxels* del tamaño especificado en base a la nube de puntos cuyo número de *voxels* será el número de cajas para esta iteración. Estos valores, tanto el tamaño de las cajas, como el número de estas se concatenan en dos listas, y por último se disminuye el tamaño conforme al decremento calculado al principio. Este proceso se repite tantas veces como número de iteraciones le hayamos indicado al método. Una vez finaliza el bucle se tiene que calcular la inversa de cada elemento de la lista que contiene los tamaños, y aplicar logaritmos en ambas listas, este paso se puede realizar perfectamente a la hora de almacenar los valores en las listas dentro del bucle, pero se ha comprobado que por cuestiones de paralelismo resulta más eficiente calcularlo aquí. Por último, sólo queda aproximar una recta a los pares de puntos obtenidos como se explica detalladamente en la Sección 5.1.5. A continuación, en el Pseudocódigo 5.1 se resume el método explicado a lo largo de este párrafo.

Código 5.1: Pseudocódigo algoritmo Box Counting

```
para i desde 0 hasta numIteraciones:
    divide figura en cajas usando tamaño_caja
    arrayNumeroCajas[i] = numeroCajasOcupadas
    arrayTamaños[i] = tamaño_Caja
    tamaño_caja = tamaño_caja - decremento
fin para
aplica logaritmo sobre arrayNumeroCajas
aplica inversa y logaritmo sobre arrayTamaños
curvaDimension = aproximaRecta(arrayNumeroCajas,arrayTamaños)
return pendiente(curvaDimensión)
```

donde *aproximaRecta* es un método que devuelve una recta aproximada a los puntos (siendo el primer parámetro los valores del eje Y y el segundo los del eje X) mediante el método de mínimos cuadrados, como vemos en la Figura 5.1, y *pendiente* es un método que devuelve la pendiente de una recta.

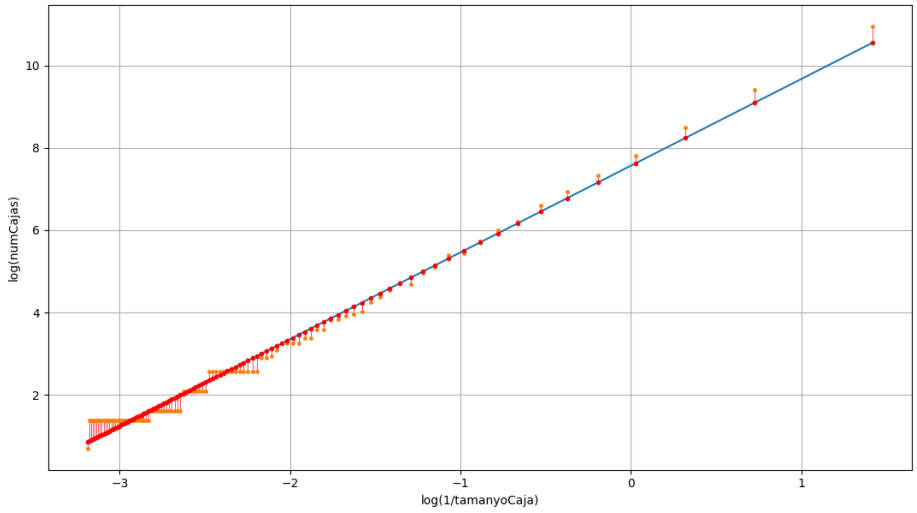


Figura 5.1: Así se genera una curva por el método de mínimos cuadrados a partir de un conjunto de puntos. En el Código 5.1 se llama a este método *aproximaRecta*.

Con este método se calcula la dimensión de conteo de cajas sobre una nube de puntos tridimensional, lo cual se puede evaluar sobre la figura completa o sobre subdivisiones de ésta. Para aplicarlo sobre fragmentos de la imagen se recorta la figura y se le pasa cada uno de estas partes al método anterior. Este proceso se conoce con el nombre de ”voxelización” y nos permite dividir la nube de puntos en *voxels* del tamaño que decidamos. En la implementación concreta de este proyecto se ha usado el método ”crop” sobre el que se hablará en la Sección 5.1.3 aunque perfectamente se podría haber realizado la misma tarea haciendo uso de los distintos métodos de la clase VoxelGrid, explicado en la Sección 5.1.4. Básicamente, el método ”crop” nos permite recortar una sección de una nube de puntos dados los dos extremos del

cuboide que cubre la sección de la nube que queremos seleccionar. El proceso para seccionar toda la figura es trivial, y consiste simplemente en partiendo del origen de ordenadas (dónde se ha situado la figura) ir aumentando las tres coordenadas en una diferencia definida por $extremoBoundingBox/numIteraciones$ para cada uno de los ejes, a través de tres bucles anidados, cada uno de ellos asociado a uno de los ejes, hasta que se ha calculado la dimensión sobre todas las secciones del modelo.

Código 5.2: Pseudocódigo algoritmo Box Counting con voxelización

```
listaSecciones = []
para cada división:
    seccion = crop(nubePuntos, división);
    listaSecciones.append(dimensionFractal(seccion, numIteraciones))
return listaSecciones
```

5.1.1 Lectura y transformación a nube de puntos

Como ya se ha explicado en la Sección 4.2, el *dataset* está en formato de mallas poligonales, pero para este trabajo se convierten a nubes de puntos. Por ello, la lectura se hace con el método "io.read_triangle_mesh" de Open3D, que permite cargar en un objeto de la clase "TriangleMesh" el contenido de ficheros con las extensiones más populares de mallas poligonales como ".ply", ".stl" o ".off" siendo esta última la que presentan los ficheros de ModelNet. Por otra parte, para convertir estas mallas a nubes de puntos primero se escalan a una dimensión similar con el método "PointCloud.scale" y se emplea el método "TriangleMesh.sample_points_uniformly" para convertir estas mallas a nubes de puntos.

5.1.2 PointCloud

Esta clase dentro del módulo "geometry" de Open3D es la encargada de proporcionar todas las funcionalidades relacionadas con las nubes de puntos. Atendiendo a la documentación oficial de la librería, esta es una clase que para representar coordenadas de puntos y opcionalmente, sus colores y los vectores normales de dichos puntos.

5.1.3 PointCloud.crop

Este es el método usado para subdividir las nubes de puntos. Nos permite dados dos puntos en el espacio "recortar" la nube de puntos acotada por la caja de recubrimiento cuyos extremos son esos dos puntos. Un ejemplo lo vemos en la Figura 5.3 donde cada *voxel* se obtiene recortando un cuboide delimitado por las líneas azules.

5.1.4 VoxelGrid

Clase de Open3D utilizada para el conteo de cajas. Nos permite agrupar los puntos de la nube en cubículos del tamaño que le indiquemos, de forma que sólo tenemos que convertir la nube de puntos a VoxelGrid y contar el número de *voxels* (cubículos) que obtenemos. Esto es lo que se hace en el Código 5.1 para calcular "numeroCajasOcupadas".

Como se puede ver en la Figura 5.4, en las primeras iteraciones no hay muchos cambios, pues en la iteración 79 sigue siendo difícil reconocer un avión, y es en las últimas 20 iteraciones

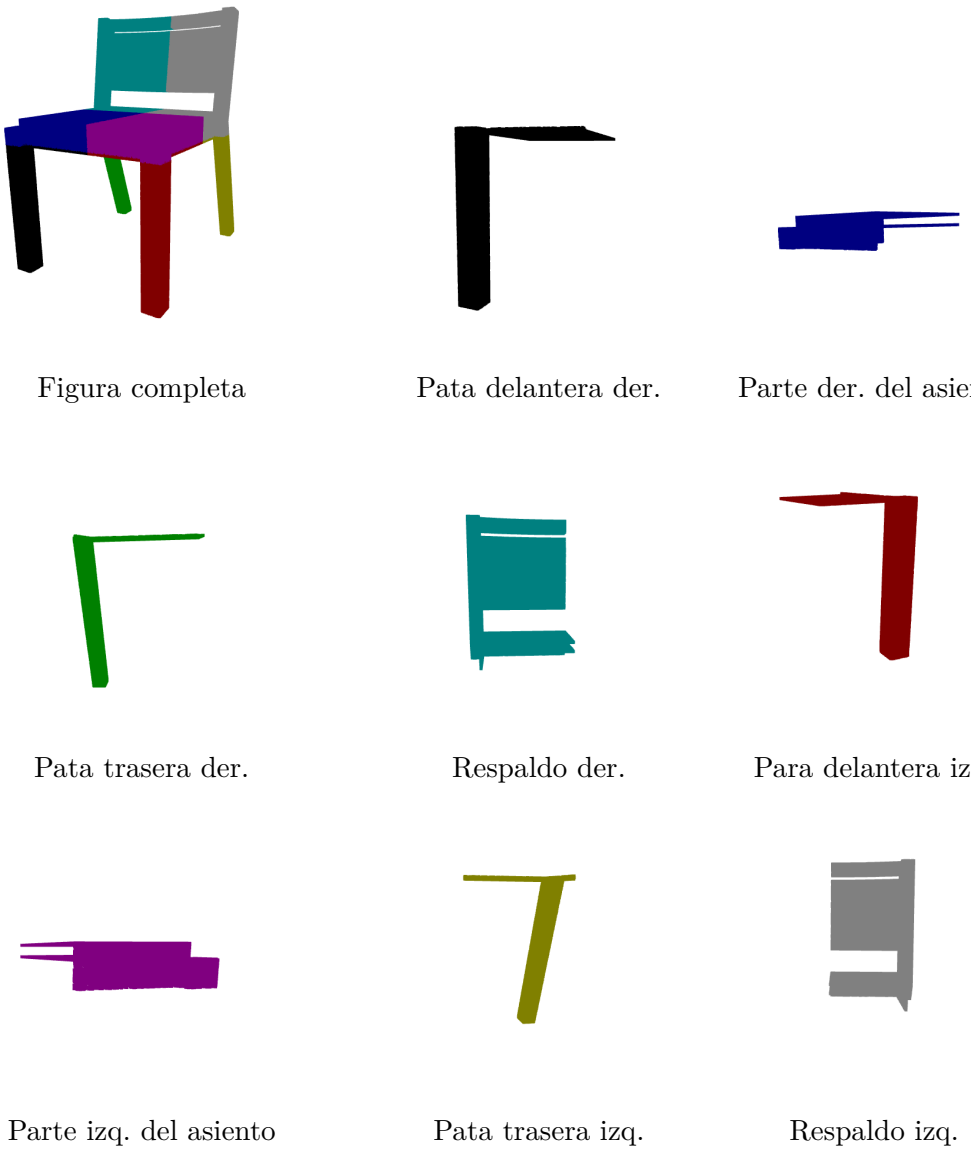


Figura 5.2: Ejemplo de cómo se divide una figura con el método *crop*.

dónde más definición gana el VoxelGrid eso explica el porqué los puntos del principio de las curvas "log-log" que se ven en este trabajo, como en la Figura 5.1 se ajusten peor a la recta que los del final de esta.

5.1.5 Ajuste de rectas

Para ajustar la rectas a un conjunto de puntos en este trabajo se ha usado la función "polyfit", de la librería NumPy (Walt y cols., 2011), que mediante el método de mínimos cuadrados, una curva de grado n a un vector de puntos. En este trabajo sólo se usará para calcular la pendiente de la curva "log-log" en el último paso del conteo de cajas, por tanto

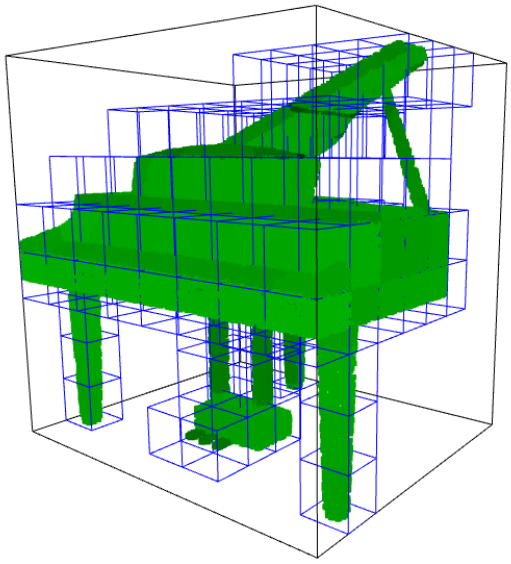


Figura 5.3: Muestra de las divisiones (en color azul) de una instancia de la clase *piano* (ModelNet40). Por otra parte, en color negro vemos la caja de recubrimiento (*bounding box)* de la figura, que es quien marca la proporción para los *voxels* de las divisiones.

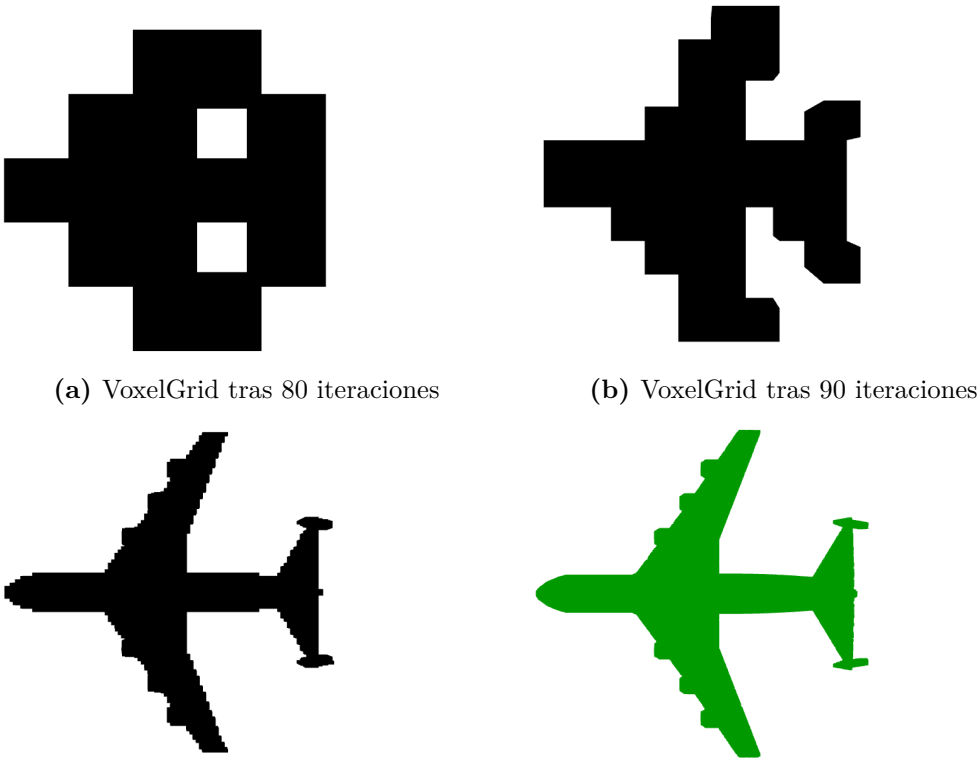
$n = 1$, pues siempre queremos ajustar una recta a los puntos. El método de mínimos cuadrados pretende encontrar, dado un conjunto de puntos, la función continua de grado n que minimice lo que se conoce como error cuadrático. Esto error se define como la suma de distancias en el eje de ordenadas entre cada punto y la recta, al cuadrado cada una de ellas para evitar valores negativos.

$$S = \sum_{i=0}^{n} d_i^2 \tag{5.1}$$

siendo en la Ecuación 5.1 que define el método de mínimos cuadrados, S el valor a minimizar, n el número de puntos, y d_i la distancia en el eje Y entre cada uno de estos puntos y la recta. En la figura 5.1 podemos ver gráficamente como funciona este método.

5.2 KNN

En este apartado por comodidad se ha implementado un clasificador KNN desde 0. El algoritmo general se puede resumir con el siguiente pseudocódigo:



(a) VoxelGrid tras 80 iteraciones

(b) VoxelGrid tras 90 iteraciones

(c) VoxelGrid tras 100 iteraciones

(d) Nube original

Figura 5.4: Evolución del VoxelGrid resultante en el conteo de cajas (para un total de 100 iteraciones).

Código 5.3: Pseudocódigo algoritmo KNN

```
para cada elementoTest de test:
    para cada elementoTrain de train:
        CalculaDistancia entre elementoTest y elementoTrain
    fin para
    Contar clases de los k elementos de train más cercanos a elementoTest
    clase de elementoTest = clase más frecuente
fin para
```

Para variar entre usar la distancia euclídea y otro tipo de distancia sólo necesitamos modificar la operación "CalculaDistancia". Algunas mejoras no funcionales respecto a este esquema básico pasan por el uso de estructuras de datos para agilizar la búsqueda de vecinos más cercanos, como podría ser un *KDtree*.

5.3 Redes Neuronales

Para las redes neuronales, se han usado dos tecnologías populares, por una parte la librería Scikit-Learn (Pedregosa y cols., 2011) y por otra el *framework* Keras (Chollet y cols., 2015). De Scikit-Learn se usó su MLP mientras que Keras se utilizó para crear una sencilla red

densamente conectada para establecer lo que se conoce como un *baseline*, un punto de partida desde el que empezar, y una CNN. En la Sección 6.3 se entrará más en detalle sobre las topologías probadas y los resultados obtenidos con cada una de ellas.

5.4 SVM

Para la clasificación con SVM se ha usado la clase "sklearn.svm.SVC" de la librería Scikit-Learn. Como ya se ha explicado en la Sección 2.2.2, esta clase implementa el enfoque uno-vs-uno para la clasificación n-aria. Sólo dos parámetros han sido alterados respecto a la configuración por defecto, los mejores resultados se han conseguido situando el valor de regularización C, con el que se debe intentar conseguir una buena generalización y que cuanto menor sea mayor será el margen entre hiperplano y vectores de soporte, en $C = 3.5$. Por otra parte, también se ha tenido que modificar el parámetro de tolerancia, es decir cómo de exigente es el criterio de parada para aceptar un hiperplano como válido, en $tol = 0.1$.

6 Experimentación

En este capítulo se expondrán todas las pruebas llevadas a cabo para validar los algoritmos implementados.

6.1 Resultados dimensión fractal

Primero se evalúa el funcionamiento del programa que calcula la dimensión de *box counting*. Como se ve en la Figura 6.1 la curva obtenida relacionando el número de cajas de cada iteración con la inversa de su lado es similar a la obtenida en trabajos previos (Ivorra Rodríguez, 2017) tanto en su forma general como en su pendiente, que al final es con lo que nos quedamos como dimensión fractal.

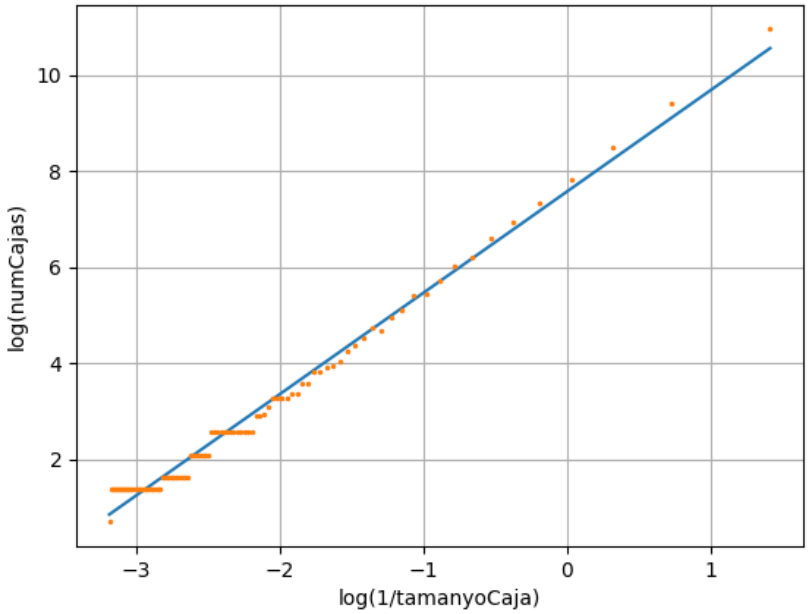
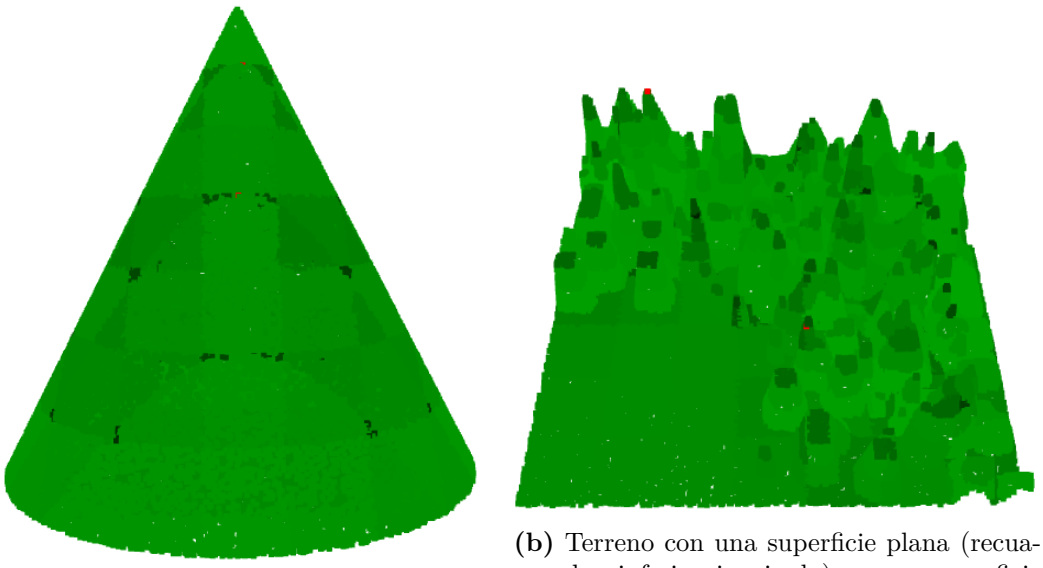


Figura 6.1: Curva log-log pirámide de Sierpinski.

Tras comprobar que el funcionamiento del método es correcto, se pasa a analizar en lo que se centra este TFG que es en su uso como descriptor 3D. Para ello, se empezará comprobando cómo se comporta esta propiedad al calcularla en dos figuras creadas por ordenador, por una parte un cono y por otra un terreno irregular sintético.

En la Figura 6.2a vemos el resultado de subdividir en *voxels* un cono y calcular la dimensión



(a) Cono.

(b) Terreno con una superficie plana (recuadro inferior izquierdo) y una superficie muy irregular (resto del terreno).

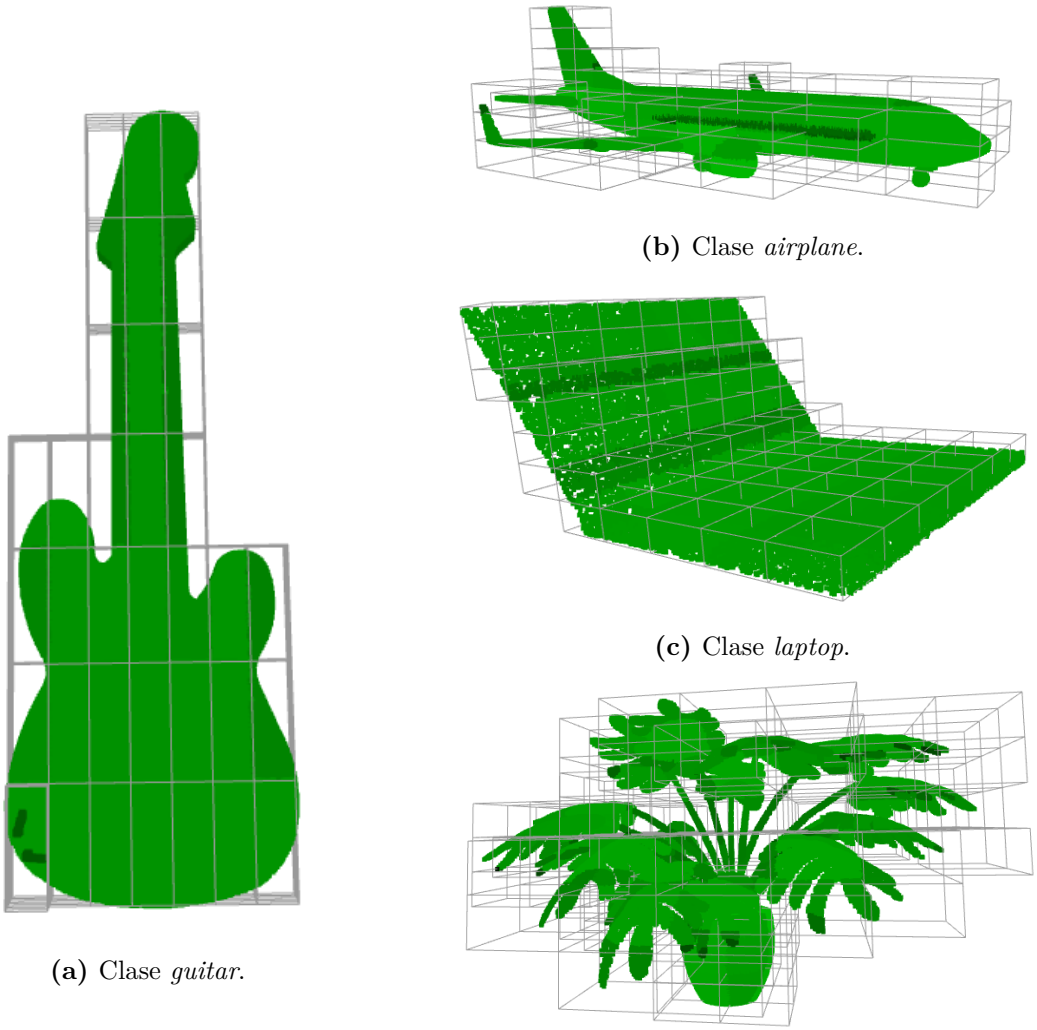
Figura 6.2: Resultado de calcular la dimensión de conteo de cajas sobre dos figuras voxelizadas, siendo los colores más claros valores mayores de dimensión fractal.

fractal de cada uno de estos *voxels*. Se pueden ver claramente las curvas cónicas (parábolas principalmente) generadas por cada *voxel* "cortando" el cono. Vemos que los valores más claros (superiores) se presentan en las zonas más grandes, mientras que los más oscuros (inferiores) se presentan en las subdivisiones que menor superficie del cono han conseguido capturar.

Por otra parte, la Figura 6.2b nos permite comparar la diferencia entre una superficie plana y una que presenta grandes irregularidades. Lógicamente, para toda la superficie regular se obtiene un resultado constante, mientras que en la irregular vemos claras diferencias. En esta, las puntas de las protuberancias presentan valores pequeños, mientras que la base de estas tiene valores similares a los de la superficie plana, y los valores más altos se registran en su parte central. Lo siguiente será analizar cómo se comporta el método con modelos reales del *dataset*.

En la Figura 6.3 vemos instancias de cuatro clases diferentes de ModelNet40. Yendo por orden, si nos fijamos en la guitarra (Figura 6.3a) vemos que lo que es el cuerpo de esta y la parte central del mástil se muestra de un color constante, pues es una superficie plana (en la mayoría de modelos de esta clase no aparecen representadas las cuerdas de la guitarra ni el puente). Este color se oscurece en los bordes del mástil y en algún borde curvo del cuerpo, pues ahí la nube llena menos su *voxel*. Las dos pequeñas zonas más oscuras que se ven en el último *voxel* inferior izquierdo son dos clavijas que son lo único presente en ese *voxel*.

Siguiendo con el modelo del avión (Figura 6.3b), destaca una franja oscura en ambos laterales de su fuselaje que se debe a una línea de *voxels* intersectando el límite del cilindro. Otros puntos que aparecen en oscuro son los extremos que se conoce como *winglets* (puntas dobladas de las alas), y un punto de las alas que intersecta mínimamente con un *voxel*. El



(a) Clase *guitar*.

(b) Clase *airplane*.

(c) Clase *laptop*.

(d) Clase *plant*.

Figura 6.3: Ejemplos de cómo quedan algunas instancias de ModelNet40 tras calcular la dimensión individualmente sobre cada uno de los *voxels* en los que estas se descomponen.

caso de la clase *laptop* (Figura 6.3c) es bastante interesante, pues como vemos el teclado al ser una superficie plana apoyada en la base de su *bounding-box* se representa con un color constante. Por otra parte, su pantalla, al estar inclinada cruza varias columnas del *voxel grid* propiciando que aparezcan unas franjas horizontales a lo largo de ella cuando el trozo de pantalla que intersecta la fila de *voxels* es pequeño. Por último, si nos fijamos en la planta (Figura 6.3d), vemos que tanto la superficie de las hojas como de la maceta aparecen en un color similar bastante claro, mientras que este color se oscurece ligeramente en los tallos y mucho más en las puntas de las hojas. Este efecto que vemos en las puntas de las hojas es idéntico al que ocurría con los *winglets* del avión, las clavijas de la guitarra y las puntas del terreno irregular en 6.2b, y creemos que puede ser uno de los factores determinantes para diferenciar los objetos. Viendo estos resultados, se pasa a evaluar si estos datos son útiles

para la clasificación de los modelos o no.

6.2 KNN

El primer clasificador usado para evaluar la viabilidad de la dimensión fractal como descriptor 3D fue un KNN previamente explicado. Se decidió comenzar con este clasificador por su simplicidad y eficiencia.

6.2.1 Dimensión de la figura completa

El primer experimento realizado consiste en calcular la dimensión fractal de la figura completa y clasificar con un KNN clásico que calcula las diferencias como la distancia euclídea.

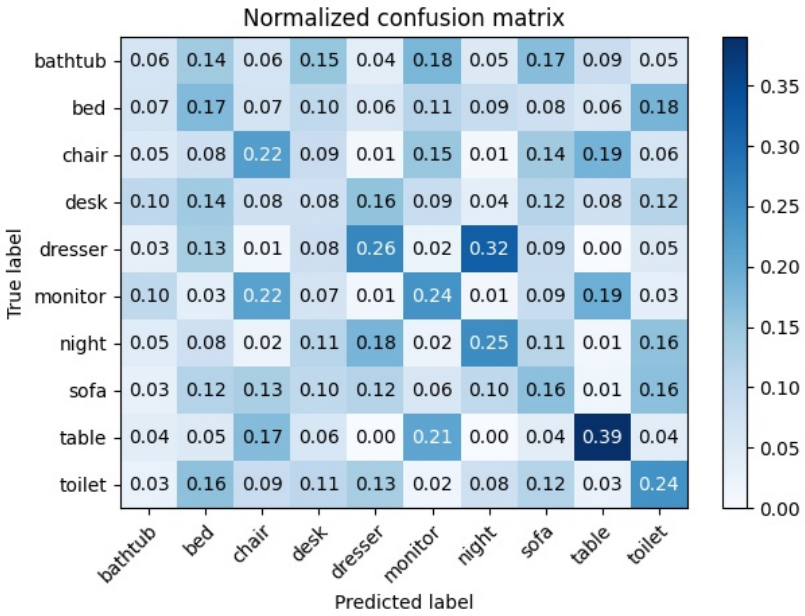


Figura 6.4: Matriz de confusión calculando la dimensión de la figura completa con $k = 5$.

Como se puede apreciar, en la Figura 6.4 el resultado de clasificar en base a la dimensión fractal de la figura completa es caótico ofreciendo un acierto escasamente superior al 20% (**21.6%**), es decir no mucho mejor que una clasificación aleatoria. Además, no denota ningún tipo de correlación entre las clases esperadas y las reales, lo cual nos indica que el método de clasificación no es válido. Esto se repite para todos los valores de k probados (1, 3, 5, 10, 25 y 45).

6.2.2 Dimensión realizando subdivisiones

Tras ver los malos resultados con la figura completa se decidió dividir el modelo en partes de igual tamaño a través del método "crop" ya explicado. Con estas divisiones cada figura deja de estar representada por un único valor real para estarlo por un vector de estos, lo

cual previsiblemente debe aportar una mejor descripción del modelo 3D. Ya se ha explicado en la Sección 5.1 cómo se genera este vector usando ”crop” y partiendo desde el origen de coordenadas avanzando hasta el eje opuesto de la caja de recubrimiento de la figura, aunque este orden es irrelevante en los clasificadores probados.

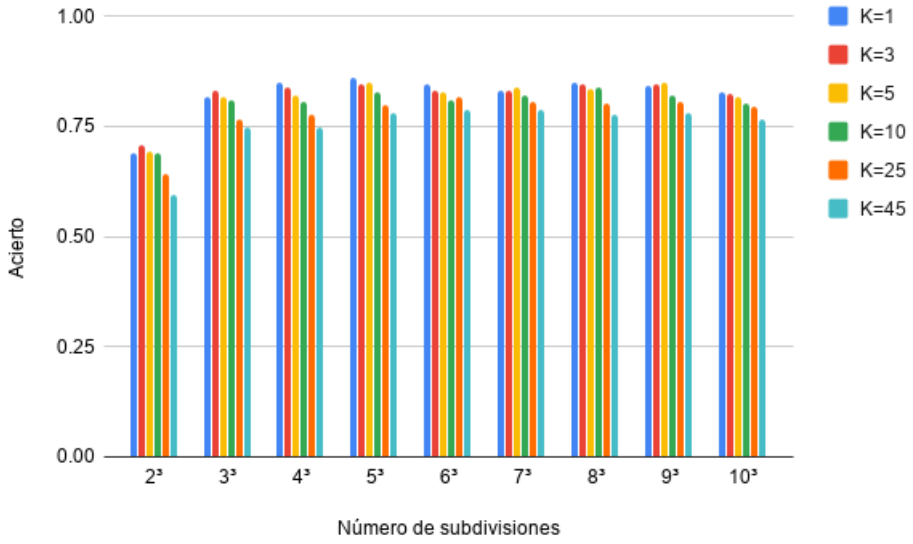


Figura 6.5: Comparativa resultados KNN para distinto número de subdivisiones y valores de K usando distancia euclídea.

6.2.2.1 Distancia euclídea

En la Figura 6.5 vemos resumido todo el banco de pruebas ejecutado con este algoritmo. Con este gráfico se quiere resumir la dependencia del acierto en dos variables, el número de partes en las que se divide cada figura (eje X) y el número de vecinos que se toma en el KNN (distintos colores de las barras). Como vemos los peores resultados aparecen con el mínimo número de subdivisiones (2^3) y forman una meseta hasta llegar al máximo (10^3) dónde vuelve a bajar el acierto. Otro elemento común es que el algoritmo funciona mejor para valores pequeños de k que para los grandes, independientemente del número de divisiones. El máximo valor conseguido en estas pruebas es **86,01%**, que se consigue con $k = 1$ y 5^3 divisiones. En la Figura 6.6 vemos su matriz de confusión. Otro hecho a destacar es que el número de subdivisiones con mejor resultado medio (para todos los valores de k) es 5^3, mientras que el valor de k con mejor resultado (para todos los números de subdivisiones) es 1. Estos datos serán la base para comparar con los próximos experimentos.

Es innegable la mejora obtenida dividiendo las figuras y clasificando en función de los vectores de dimensión fractal donde cada elemento es una de las partes del modelo.

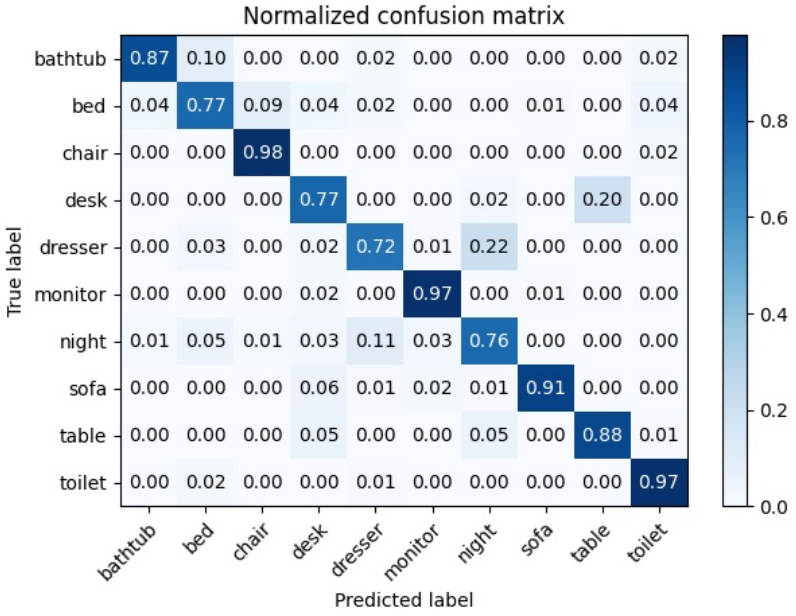


Figura 6.6: Matriz de confusión realizando 5^3 subdivisiones usando KNN con $k = 1$ y distancia euclídea. Se obtiene una precisión del 86,01%

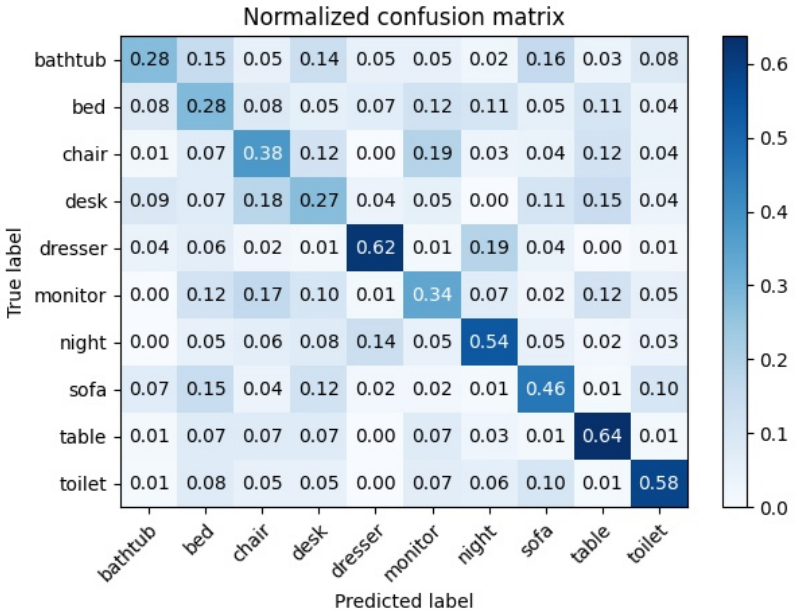


Figura 6.7: Matriz de confusión resultante de clasificar haciendo 5^3 subdivisiones de la figura, calculando su histograma y tomando $k = 1$ en el KNN con lo que se obtiene una precisión del 44,38%.

6.2.2.2 Histogramas y su uso en clasificación

El siguiente paso fue analizar los histogramas resultantes del algoritmo de *box counting*. Estos histogramas lo que nos denotan es la frecuencia con la que se repite un valor de dimensión fractal entre todas las subdivisiones de la figura. En la clasificación de imágenes, un método muy común que ofrece frecuentemente buenos resultados consiste en calcular el histograma de alguna propiedad de la imagen (como podría ser el color) y usar estos valores del histograma como entrada a un clasificador (comúnmente del tipo CNN o SVM) sin usar la imagen como tal.

Siguiendo la idea de clasificación basada en histogramas y a la vez con la intención de indagar un poco más sobre el comportamiento de la dimensión fractal, se replica el mejor experimento de la Sección 6.2.2.1 donde se dividía en 5^3 secciones cada figura y se tomaba $k = 1$ como valor de vecindad en el clasificador KNN. En la Figura 6.7 podemos apreciar el mal resultado obtenido esta vez, alcanzando sólo un 44,38% de precisión, empeorando algo más de 31 puntos porcentuales con respecto a la clasificación sin usar histogramas.

El experimento se repite con $k = 5$ y manteniendo el resto de parámetros, algo que en la versión sin histogramas había resultado en un 84,8% de acierto. En esta ocasión, con histogramas, se queda en un escaso 48,34% que pese a ser cuatro puntos mayor que cogiendo $k = 1$ sigue siendo un resultado muy negativo y lo que es peor, la tendencia alcista no se mantiene con valores de k superiores.

Por último, sospechando que la baja precisión pudiera estar relacionada con la escasa precisión de los histogramas, que hasta ahora se habían realizado tomando 50 intervalos, se decide probar con 20 y 100 intervalos, para ver el efecto de esta variable sobre el resultado. No se superan los 100 intervalos porque ya en esos valores la frecuencia de cada intervalo es muy reducida, como vemos en la Figura 6.8b.

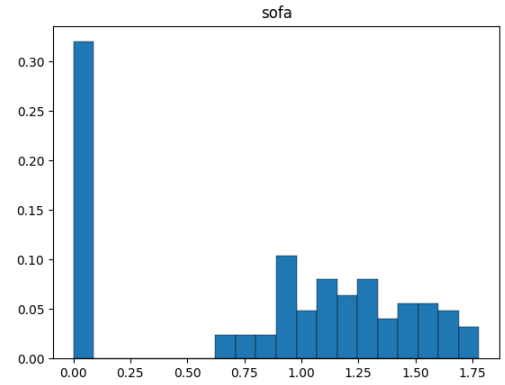
Como vemos en las Figuras 6.8c y 6.8d el resultado dividiendo los histogramas con distinto número de intervalos sigue siendo muy pobre, por esto se concluye que se debe volver a la clasificación basada directamente en los valores de dimensión de conteo de cajas.

Analizando los histogramas resultantes, como los que podemos ver en la Figura 6.9, llama la atención la destacada presencia del valor 0. Esto ocurre porque este es el valor asignado en mi implementación del método de conteo de cajas para aquellas regiones vacías (sin ningún punto) del modelo. Al sospechar que esto nos puede estar haciendo perder información, provocando que en el clasificador pesen más el número de regiones vacías que la propia dimensión, se decide implementar una versión del clasificador KNN que use la distancia de Mahalanobis (o distancia euclídea estandarizada, que en este caso son equivalentes), explicada en la Sección 2.2.1.2, en lugar de la euclídea como medida de diferencia entre instancias.

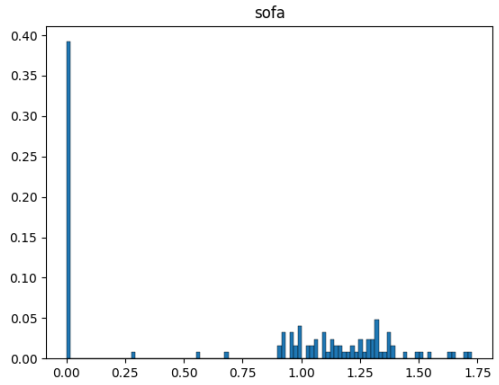
6.2.2.3 Distancia de Mahalanobis

Para averiguar si realmente la dimensión fractal está siendo lo que determine la clasificación, repetimos los experimentos previos con esta medida.

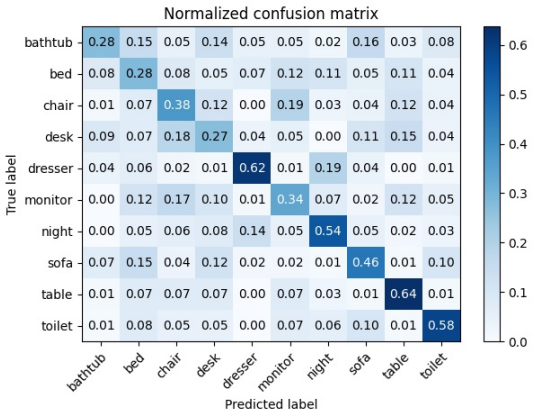
Como podemos ver en la Figura 6.10 los resultados son muy similares a la gráfica usando distancia euclídea como medida de diferencia en el algoritmo KNN (Figura 6.5). Se vuelve a repetir la estructura general con valores bajos en 2^3 particiones que se mantienen rondando el 86% hasta llegar a 10^3, dónde vuelven a decaer. También se repite el mejor funcionamiento para valores bajos de k, siendo $k = 1$ el valor con mejor acierto medio de nuevo. Por otra



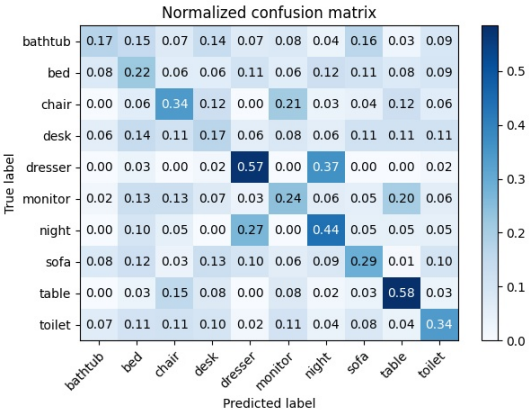
(a) Histograma de 20 intervalos.



(b) Histograma de 100 intervalos.



(c) Clasificando con histogramas de 20 intervalos.

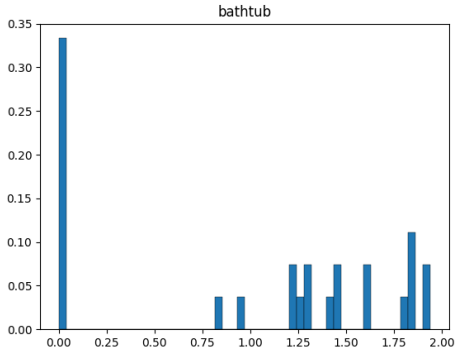


(d) Clasificando con histogramas de 100 intervalos.

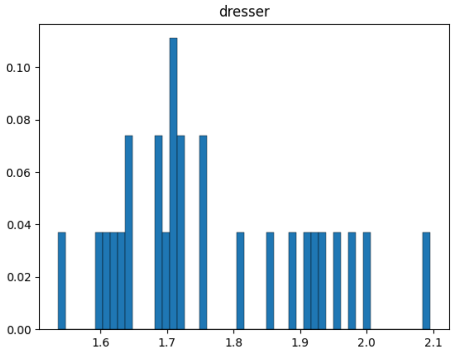
Figura 6.8: Matrices de confusión obtenidas clasificando con histogramas de 20 y 100 intervalos en la fila inferior y dos ejemplos de estos histogramas en la fila superior. Los resultados son respectivamente del 44,38% y 31.83% de acierto muy por debajo de lo conseguido sin histogramas.

parte, como elementos diferenciativos, la combinación de máximo acierto ahora se encuentra tomando 8^3 divisiones de la figura y $k = 1$, donde se obtiene un **87,00%** de precisión (un punto más que con la distancia euclídea). Además, mientras que en la distancia euclídea el número de particiones con mayor precisión media (para los seis valores de k) era 5^3, ahora usando la distancia de Mahalanobis se ha desplazado hasta 8^3 con una precisión media del 83,99% frente al 82,71% de la distancia euclídea. Como último dato a comparar, la precisión media para todas las pruebas aumenta de un 81,53% a un 82.48% con esta nueva medida de diferencia entre vectores.

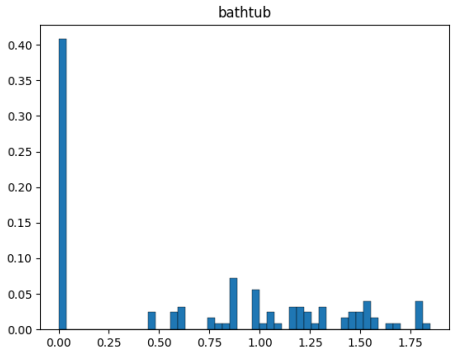
¿Qué se puede extraer de estos datos? Todos ellos denotan escasa variabilidad entre los dos métodos (de entre uno y dos puntos porcentuales en la mayoría de pruebas), a la vez que una ligera mejora en líneas generales (tanto en la mejor instancia como en la media). Esto nos indica que efectivamente, los resultados expuestos en la Sección 6.2.2.1 realmente denotan



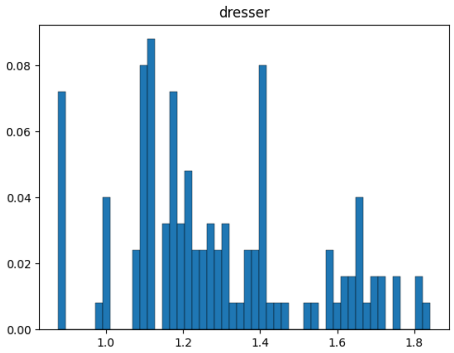
3^3 divisiones en *bathtub*



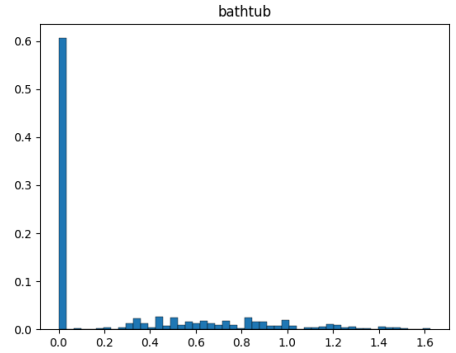
3^3 divisiones en *dresser*



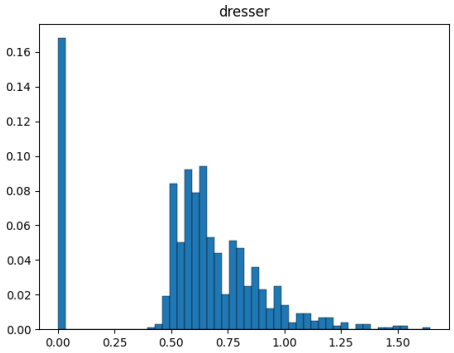
5^3 divisiones en *bathtub*



5^3 divisiones en *dresser*



10^3 divisiones en *bathtub*



10^3 divisiones en *dresser*

Figura 6.9: Comparativa de los histogramas resultantes para algunas figuras del conjunto de entrenamiento en función del número de divisiones que tomamos en la figura.

que la dimensión fractal está funcionando adecuadamente como descriptor, ya que de no ser así, al perder importancia los ceros en la distancia de Mahalanobis, los resultados con esa medida deberían ser sensiblemente inferiores.

Otro hecho que podemos resaltar es que las mejoras aparecen principalmente para núme-

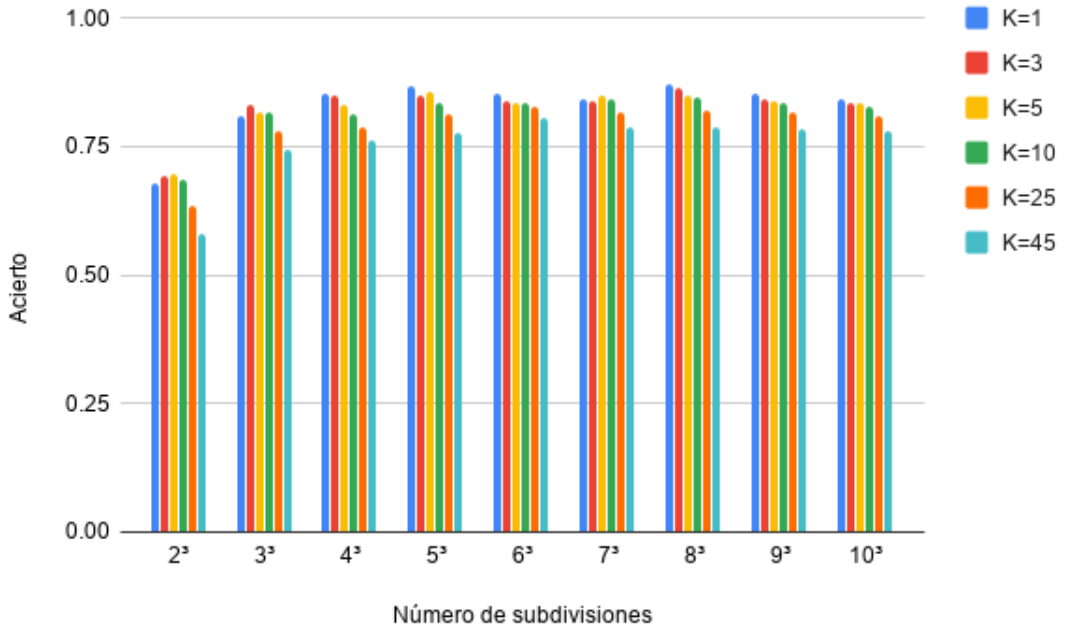


Figura 6.10: Comparativa resultados KNN para distinto número de subdivisiones y valores de K usando distancia de Mahalanobis.

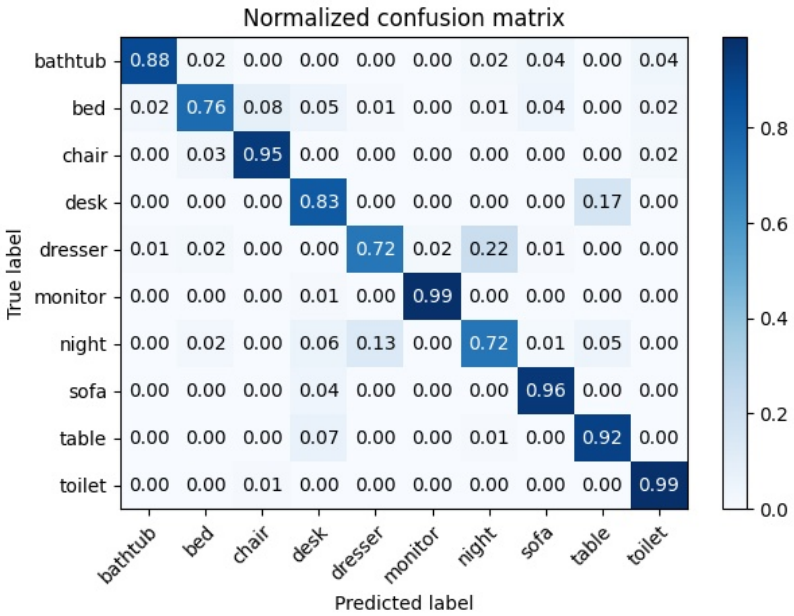


Figura 6.11: Matriz de confusión realizando 8^3 subdivisiones usando KNN con $k = 1$ y midiendo con distancia de Mahalanobis. Se obtiene una precisión del **87,00%**.

ros altos de subdivisiones de la figura. Este hecho, junto con lo ya mencionado en el párrafo anterior puede indicar que los ceros perjudican a la clasificación, pues es en valores altos del número de divisiones dónde lógicamente aparecen más ceros y al ignorar estos con la distancia de Mahalanobis son los valores para los que más ha mejorado la clasificación. Todo esto nos indica por tanto que efectivamente la dimensión de conteo de cajas se está comportando adecuadamente como descriptor. Una vez tenemos clara la viabilidad del uso de esta propiedad, es momento de pasar a clasificadores más complejos para ver hasta dónde puede llegar su precisión.

6.3 Redes neuronales

El siguiente clasificador a evaluar son las redes neuronales, comenzando por un MLP, siguiendo por redes densamente conexas y por último probando una pequeña CNN.

6.3.1 Perceptrón multicapa

La primera prueba se realiza con un MLP, uno de los tipos de red más sencillos que existen, para así tener una primera medida de si estos clasificadores pueden funcionar. Se comienza por esta prueba porque requiere cambios casi insignificantes sobre el código existente. Con una topología de tres capas, y sólo 10 neuronas en la capa oculta y dividiendo en 3^8 partes los modelos se consigue sólo un 79,73% de acierto, por tanto se decide pasar a probar redes neuronales estándar y desechar esta alternativa al ver que no se mejoran los resultados en ningún caso.

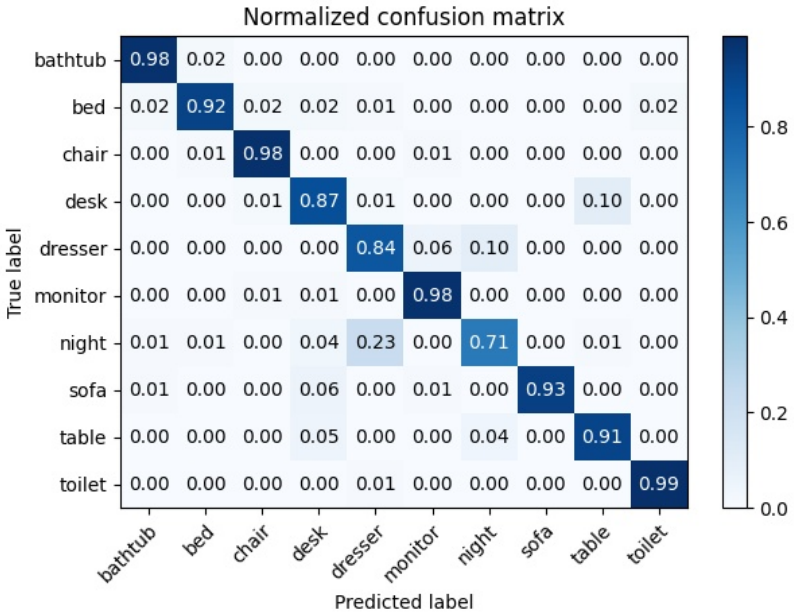


Figura 6.12: Matriz de confusión realizando 9^3 subdivisiones usando red densamente conexa. Se obtiene una precisión del **90,75%**.

6.3.2 Redes densamente conectadas

Para comenzar la clasificación con redes neuronales se prueba a usar lo que popularmente se conoce como una *baseline*, es decir una topología de red relativamente ligera que nos de un primera impresión de si esta está aprendiendo correctamente. Se empieza con redes sencillas para evitar problemas como el *overfitting*, que se da cuando nuestra red tiene "demasiada" capacidad y en lugar de aprender memoriza las instancias de entrenamiento dando pobres resultados cuando esta se traslada el entorno de *test*. Este primera arquitectura está compuesta de sólo dos capas, una de entrada con tantas neuronas como características tienen los vectores de entrada (es decir, el número de divisiones que hacemos de la figura) con activación sigmoidea y una capa de salida, con 10 neuronas (una por cada categoría) que usan la función *softmax*.

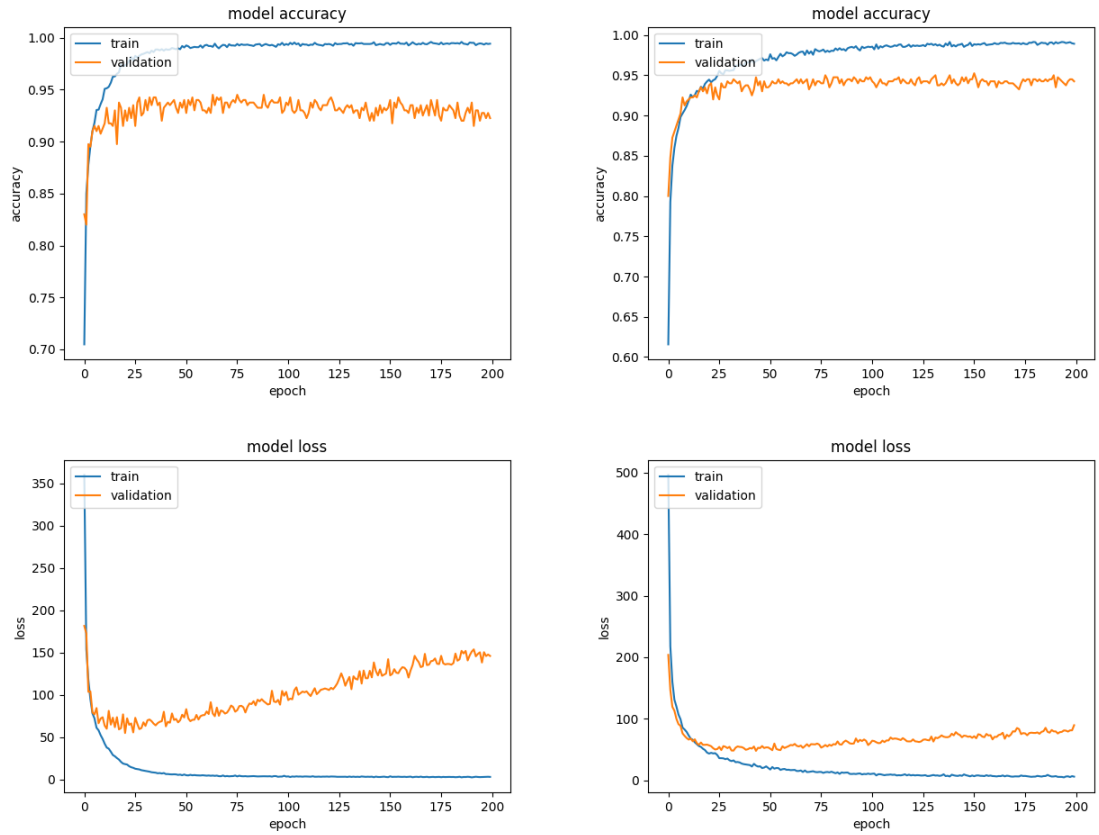


Figura 6.13: Historiales de *accuracy* y *loss* antes (columna izquierda) y después (columna derecha) de aplicar Dropout.

Repetimos los experimentos que mejor han funcionado con KNN, es decir, usando 5^3 y 8^3 subdivisiones en cada imagen y obtenemos como resultado precisiones del **88,33%** y **89,87%**. Valorando muy positivamente estos resultados, ampliamos la prueba para todo el intervalo de subdivisiones para el que hemos probado KNN (intervalo [2,10]). Los mejores resultados en este banco de pruebas se obtienen con 9^3 subdivisiones, con un acierto del **90,75%**. En

la Figura 6.12 vemos la matriz de confusión de este experimento y en la primera columna de la Figura 6.13 encontramos los historiales de entrenamiento de la red. Como se puede ver en estos historiales, pese a la sencillez de la red aparentemente hay *overfitting* pues presentan la apariencia típica de las redes con este problema, donde los resultados en el conjunto de entrenamiento mejoran firmemente junto con los de validación durante las primeras iteraciones, pero ambos valores se separan pronto, quedando muy distanciados ambos resultados. Para contrarrestar esto se utilizan técnicas de regularización, y en este caso de ha aplicado una conocida como Dropout (Srivastava y cols., 2014). Es una técnica muy sencilla que consiste en "apagar" (poner sus pesos a 1 para que no cuenten) aleatoriamente algunas neuronas para forzar a la red a aprender en lugar de memorizar. Con esta técnica se consigue una muy sensible mejora hasta el **90,97%** de acierto, y como vemos en la segunda columna de la Figura 6.13, las gráficas de acierto y pérdida son mucho más correctas, disminuyendo la distancia entre validación y entrenamiento.

6.3.3 Redes convolucionales

Siguiendo con las redes neuronales, el último experimento ha consistido en aplicar una convolución 3D sobre los datos de la dimensión fractal. Desafortunadamente, no se consiguen mejorar los resultados con este método pues el mejor resultado conseguido con él se estanca en un 88,99%, usando 9^3 divisiones de la figura.

6.4 SVM

El último de los clasificadores a probar será del tipo SVM.

Con este método se obtienen los mejores resultados hasta el momento, llegando a un **92,84%** de precisión en ModelNet10 obteniendo la matriz de confusión que se muestra en la Figura 6.14. Además, si nos fijamos en los resultados vemos que los principales errores se dan entre clases cuya conflictividad ya se ha explicado en la Sección 4.2.1. Este hecho, más que hablar de un mal funcionamiento del descriptor (lo cual por supuesto tampoco es descartable) nos indica un correcto funcionamiento de este al errar en categorías semánticamente similares y que como ya se ha explicado anteriormente, en algunas ocasiones no son diferenciables.

6.4.1 ModelNet40

Viendo el éxito de este método de clasificación, se decide saltar a un problema de mayor envergadura, pasando de usar como *benchmark* ModelNet10 a ModelNet40. No se ha probado este *dataset* antes porque se consideraba más importante encontrar primero el mejor método, ya que, como se ha explicado en la Sección 4.2.2 ModelNet40 es un *dataset* mucho más grande, lo cual ralentiza significativamente las pruebas, retrasando la toma de decisiones para avanzar en el desarrollo.

Con SVM para ModelNet40 con la configuración ya explicada, se consigue un acierto del **88,74%** con 7^3 divisiones en la figura.

Con estos resultado, si vamos a la ModelNet Benchmark Leaderboard y nos fijamos en cada uno de los métodos, vemos que hay un total de 26 algoritmos propuestos que usen datos 3D

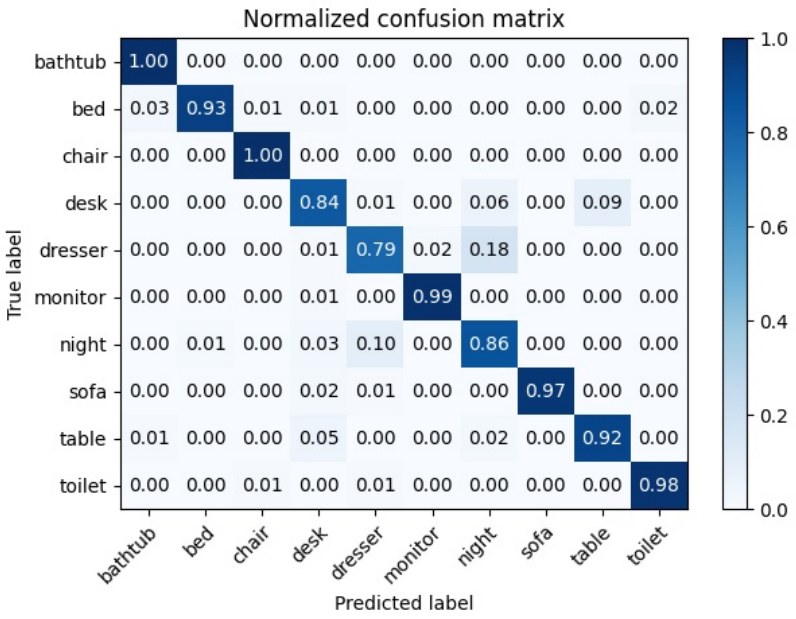


Figura 6.14: Matriz de confusión realizando 5^3 subdivisiones usando SVM. Se logra una precisión del **92,84%**.

sin realizar proyecciones previas, que lógicamente son aquellos con los que debemos comparar con el presentado en este TFG. Viendo los resultados reportados por los distintos artículos, este trabajo se situaría en la **posición 12ª** (de 26) para ModelNet40 y en la **posición 13ª** (de 26) para ModelNet10.

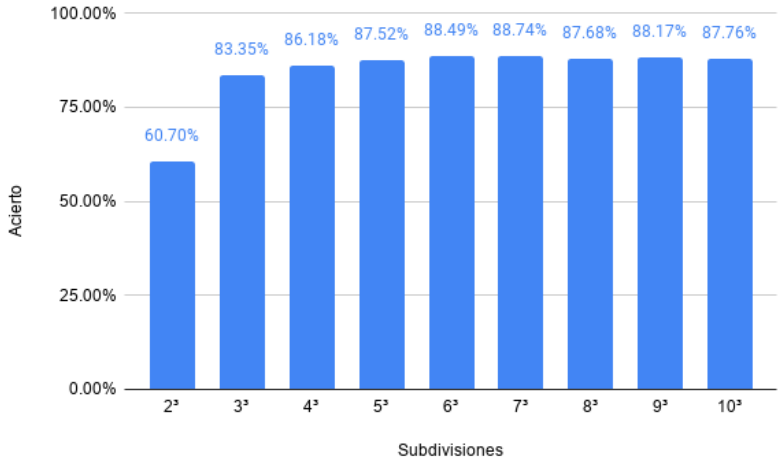


Figura 6.15: Porcentaje de acierto en ModelNet40 en función del número de partes en las que se seccione la figura.

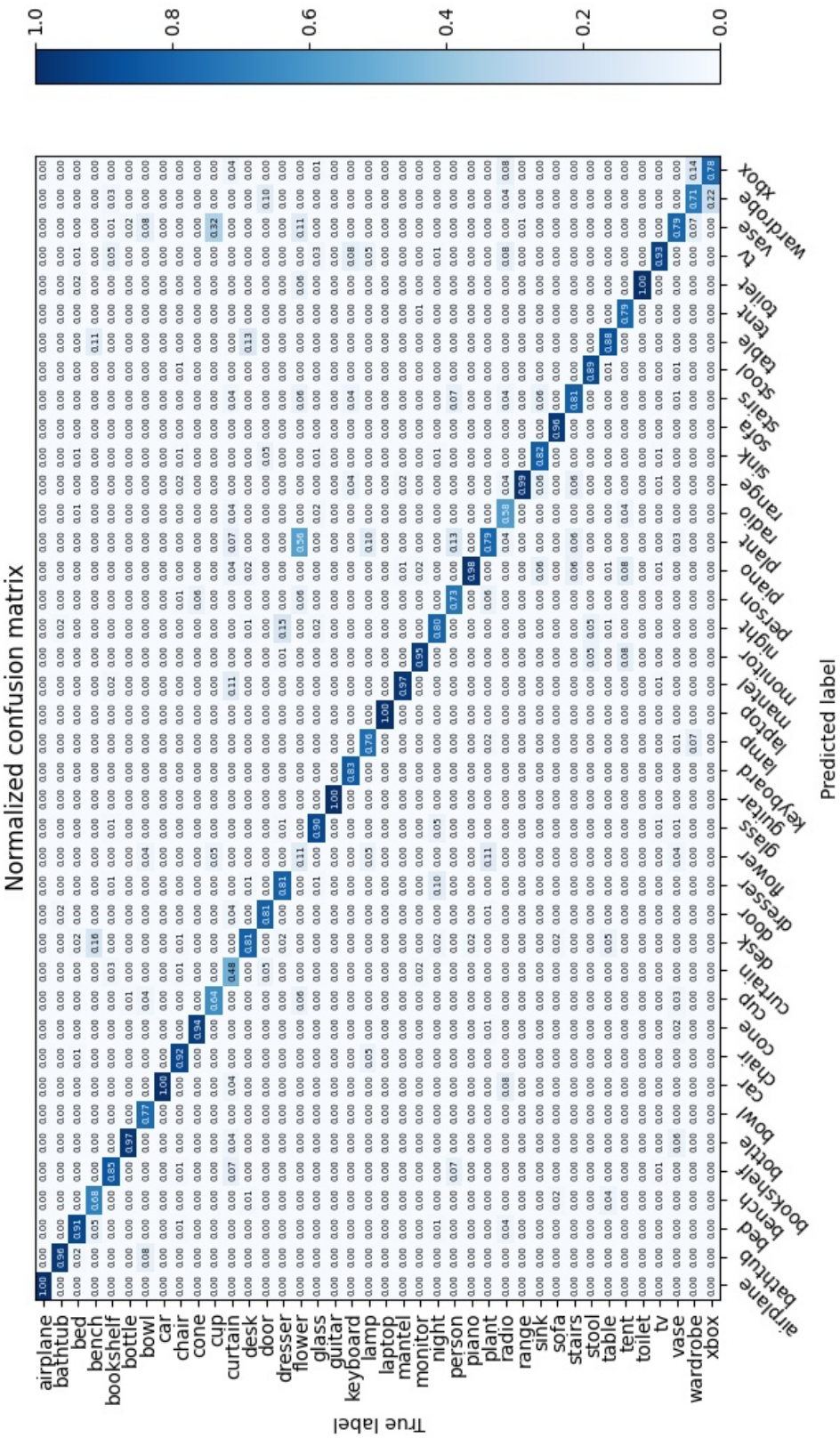


Figura 6.16: Matriz de confusión de ModelNet realizando 7^3 subdivisiones clasificando con SVM. Se logra una precisión del 88,74%.

7 Conclusiones

En este último capítulo se expondrán algunas de las conclusiones obtenidas en base a la realización de este trabajo.

La primera conclusión a exponer es que, pese al ingente volumen de la producción científica surgido en los últimos años en el ámbito de la visión artificial (algunos ejemplos de ella se han visto en la Sección 2.4), esta es una disciplina dónde aún queda mucho por explorar.

Respecto a los métodos de reconocimiento de nubes de puntos puramente tridimensionales, si bien en los últimos años se han dejado un poco de lado frente a aquellos que realizan proyecciones a dos dimensiones, siguen siendo una alternativa muy a valorar, donde aún queda mucho por investigar, como el uso de nuevos detectores y descriptores.

Tomando como referencia un trabajo anterior (Ivorra Rodríguez, 2017) en lenguaje C++ y usando otras librerías para procesado de nubes de puntos, se ha conseguido adaptar el método de conteo de cajas al lenguaje Python con la librería Open3D para el manejo de nubes.

Por último, la principal conclusión es que la viabilidad del uso de la dimensión de conteo de cajas como descriptor global ha quedado acreditada a través de la experimentación expuesta en el Capítulo 6, por tanto, se puede afirmar que se ha cumplido el objetivo principal de este proyecto. Además, los resultados conseguidos en el problema de clasificación de objetos usando ModelNet10 y ModelNet40 como *benchmarks* han sido muy satisfactorios, consiguiendo un acierto del 92,84% y el 88,74% respectivamente, lo cual sitúa este proyecto en las posiciones 13ª y 12ª con mayor acierto entre los algoritmos que usan datos 3D sin proyecciones en sendos *datasets*.

7.1 Trabajos futuros

A lo largo del desarrollo de este TFG se han planteado algunas posibles mejoras que, o bien por cuestiones de limitación temporal por los plazos de entrega, o bien por escapar al ámbito de este proyecto, no han podido ser probadas, podría ser interesante abordarlas en proyectos futuros.

Por una parte, en el Capítulo 2 se ha hablado de algunos de los muchos métodos que existen para calcular la dimensión fractal. Parece lógico que si el método de conteo de cajas ha funcionado para la clasificación de objetos, otros métodos podrían también funcionar y quizá mejorar los resultados o la eficiencia. Otra vía de exploración sería usar la dimensión fractal, calculada por el método apropiado, para describir otros objetos matemáticos, como se ha visto en algunos de los proyectos de la Sección 2.4.

Otro camino a explorar son algunas mejoras en la implementación del método de *box counting*, como por ejemplo tomar un subconjunto de los puntos para aproximar la recta cuya pendiente nos dice la dimensión, en lugar de aplicar mínimos cuadrados sobre todos los puntos. Con esto quizá se evitaría la distorsión que producen en la recta los puntos más a la izquierda de la gráfica, como se ve en la Figura 5.1. En cuanto a eficiencia, no se ha abordado

el tema de la paralelización masiva porque de momento Open3D no está optimizado para, por ejemplo, ejecutar en unidades de procesamiento gráfico, como sí ocurre en otros *frameworks*.

Bibliografía

Bebis, G., Boyle, R., Parvin, B., Koracin, D., Porikli, F., Skaff, S., … others (2016). *Advances in visual computing: 12th international symposium, isvc 2016, las vegas, nv, usa, december 12-14, 2016, proceedings* (Vol. 10073). Springer.

Bonev, B., Petkov, P., y Spassova, N. (s.f.). Modified fractal super j-pole antenna.

Bruno, O. M., de Oliveira Plotze, R., Falvo, M., y de Castro, M. (2008). Fractal dimension applied to plant identification. *Information Sciences*, *178*(12), 2722–2733.

Chatfield, K., Simonyan, K., Vedaldi, A., y Zisserman, A. (2014). Return of the devil in the details: Delving deep into convolutional nets. *arXiv preprint arXiv:1405.3531*.

Chollet, F., y cols. (2015). *Keras.* `https://keras.io`.

Garcia-Garcia, A., Gomez-Donoso, F., Garcia-Rodriguez, J., Orts-Escolano, S., Cazorla, M., y Azorin-Lopez, J. (2016). Pointnet: A 3d convolutional neural network for real-time object class recognition. En *2016 international joint conference on neural networks (ijcnn)* (pp. 1578–1584).

Gómez, C., Mediavilla, Á., Hornero, R., Abásolo, D., y Fernández, A. (2009). Use of the higuchi's fractal dimension for the analysis of meg recordings from alzheimer's disease patients. *Medical engineering & physics*, *31*(3), 306–313.

Gomez-Donoso, F., Garcia-Garcia, A., s. Orts-Escolano, Garcia-Rodriguez, J., y Cazorla, M. (2017, May). Lonchanet: A sliced-based cnn architecture for real-time 3d object recognition. En *2017 international joint conference on neural networks (ijcnn)*.

Hänsch, R., Weber, T., y Hellwich, O. (2014). Comparison of 3d interest point detectors and descriptors for point cloud fusion. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *2*(3), 57.

Hausdorff, F. (1918). Dimension und äußeres maß. *Mathematische Annalen*, *79*(1-2), 157–179.

He, K., Zhang, X., Ren, S., y Sun, J. (2016). Deep residual learning for image recognition. En *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 770–778).

Higuchi, T. (1988). Approach to an irregular time series on the basis of the fractal theory. *Physica D: Nonlinear Phenomena*, *31*(2), 277–283.

Hoang, L., Lee, S.-H., Kwon, O.-H., y Kwon, K.-R. (2019). A deep learning method for 3d object classification using the wave kernel signature and a center point of the 3d-triangle mesh. *Electronics*, *8*(10), 1196.

Ivorra Rodríguez, G. (2017). Cálculo de la dimensión fractal sobre objetos 3d.

Kanezaki, A., Matsushita, Y., y Nishida, Y. (2016). *Rotationnet: Joint object categorization and pose estimation using multiviews from unsupervised viewpoints.*

Knerr, S., Personnaz, L., y Dreyfus, G. (1990). Single-layer learning revisited: a stepwise procedure for building and training a neural network. En *Neurocomputing* (pp. 41–50). Springer.

Koch, H. (1904). Sur une courbe continue sans tangente, obtenue par une construction géométrique élémentaire. *Arkiv for Matematik, Astronomi och Fysik*, *1*, 681–704.

Krizhevsky, A., Sutskever, I., y Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. En *Advances in neural information processing systems* (pp. 1097–1105).

Lahiri, S., y Ghanta, K. (2009). Artificial neural network model with the parameter tuning assisted by a differential evolution technique: The study of the hold up of the slurry flow in a pipeline. *Chemical Industry and Chemical Engineering Quarterly/CICEQ*, *15*(2), 103–117.

Li, J., Chen, B. M., y Hee Lee, G. (2018). So-net: Self-organizing network for point cloud analysis. En *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 9397–9406).

Mandelbrot, B. (1993). How long is the coast of britain? statistical self–similarity and fractional dimension. *Classics on fractals, edited by GE Edgar*, 351–358.

Mandelbrot, B. B. (1983). *The fractal geometry of nature* (Vol. 173). WH freeman New York.

Marton, Z.-C., Pangercic, D., y Beetz, M. (2011). *Efficient surface and feature estimation in rgbd.* Descargado de `https://ias.in.tum.de/_media/events/rgbd2011/marton.pdf`

Marton, Z.-C., Pangercic, D., Blodow, N., Kleinehellefort, J., y Beetz, M. (2010). General 3d modelling of novel objects from a single view. En *2010 ieee/rsj international conference on intelligent robots and systems* (pp. 3700–3705).

McCulloch, W. S., y Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115–133.

Ng, A., Katanforoosh, K., y Bensouda Mourri, Y. (s.f.). *Convolutional neural networks.* Descargado 2020-05-25, de `https://www.coursera.org/learn/convolutional-neural-networks`

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., … Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Ralhan, A. (2018, feb). *Self organizing maps.* Descargado de `https://towardsdatascience.com/self-organizing-maps-ff5853a118d4`

Rian, I. M., y Sassone, M. (2014). Fractal-based generative design of structural trusses using iterated function system. *International Journal of Space Structures*, *29*(4), 181–203.

Robinson, J. C. (2010). *Dimensions, embeddings, and attractors* (Vol. 186). Cambridge University Press.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, *65*(6), 386.

Shannon, C. E. (1948). A mathematical theory of communication. *Bell system technical journal*, *27*(3), 379–423.

Shao, B. (2016). Pericardium segmentation in non-contrast cardiac ct images using convolutional neural networks..

Shen, G. (2002). Fractal dimension and fractal growth of urbanized areas. *International Journal of Geographical Information Science*, *16*(5), 419–437.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., y Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, *15*(1), 1929–1958.

Suratkar, C. S. (2020, Apr). A review on big data mining in healthcare. *International Research Journal of Engineering and Technology (IRJET)*, *07*(04), 5231–5231. Descargado de `https://www.irjet.net/archives/V7/i4/IRJET-V7I4985.pdf`

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., … Rabinovich, A. (2015). Going deeper with convolutions. En *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 1–9).

Thäle, C., y Freiberg, U. (2008). A markov chain algorithm for determining crossing times through nested graphs. *Discrete Mathematics & Theoretical Computer Science*.

Ulaş, Ç. (2013). *Incorporation of a language model into a brain computer interface based speller* (Tesis Doctoral no publicada).

Walt, S. v. d., Colbert, S. C., y Varoquaux, G. (2011). The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, *13*(2), 22–30.

Wu, Z., Song, S., Khosla, A., Yu, F., Zhang, L., Tang, X., y Xiao, J. (2015). 3d shapenets: A deep representation for volumetric shapes. En *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 1912–1920).

Zhou, Q.-Y., Park, J., y Koltun, V. (2018). Open3d: A modern library for 3d data processing. *arXiv preprint arXiv:1801.09847*.

Lista de Acrónimos y Abreviaturas

2D	Dos Dimensiones.
3D	Tres Dimensiones.
CNN	Convolutional Neural Network (red neuronal convolucional).
GAN	Generative Adversarial Networks.
KNN	K-Nearest Neighbours.
MEG	Magnetoencefalograma.
MLP	Multi-Layer Perceptron (perceptrón multicapa).
OFF	Object File Format.
RSD	Radius-based Surface Descriptor.
SOM	Self-Organizing Map.
SVM	Support Vector Machines.
TFG	Trabajo Final de Grado.